2015

Motion control using optical flow of sparse image features

Seebacher, J. Paul

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

MOTION CONTROL USING OPTICAL FLOW OF SPARSE IMAGE FEATURES

by

J. PAUL SEEBACHER

B.A., B.E., Dartmouth College, 2011

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2015

Approved by

First Reader

John Baillieul, Ph.D.
Professor of Systems Engineering,
Professor of Mechanical Engineering,
Professor of Electrical and Computer Engineering

Second Reader

Hua Wang, Ph.D.
Associate Professor of Systems Engineering,
Associate Professor of Mechanical Engineering

Third Reader

Mac Schwager, Ph.D.
Assistant Professor of Systems Engineering,
Assistant Professor of Mechanical Engineering

Ozymandias

By Percy Bysshe Shelley

I met a traveller from an antique land
Who said: "*Two vast and trunkless legs of stone*
Stand in the desert. Near them, on the sand,
Half sunk, a shattered visage lies, whose frown,
And wrinkled lip, and sneer of cold command,
Tell that its sculptor well those passions read
Which yet survive, stamped on these lifeless things,
The hand that mocked them and the heart that fed
And on the pedestal these words appear:
My name is Ozymandias, king of kings
Look on my works, ye Mighty, and despair!
Nothing beside remains. Round the decay
Of that colossal wreck, boundless and bare
The lone and level sands stretch far away."

Acknowledgments

Distant goals entice and excite; delving into such goals quickly demonstrates how one may become captivated by innovation, embrittled by failures, emboldened by realizations, tenacious in the face of unrelenting toil, and singularly lost in the pursuit of a goal. Passion is a wonderful and terrible thing. But occasionally, after perpetual bouts with masochism, on the verge of consummation, one's head emerges from below the water for a single instant, a crisp breath, and clear view of the world around him. In that instant, is recognition. The effort, the burden, the joys, the mistakes, the time, the waste, the successes, and the failures. But most importantly among these findings, behind these goals, rarely recognized yet absolutely imperative, is the support. The unflinching and unwavering support. Selflessness in the face of selfish obsession. Sanity.

I would like to thank my thesis advisor, Professor Baillieul, for his unwavering patience, clear guidance and support. His instruction has illuminated the way throughout my time at Boston University. Additionally I would like to thank Professor Wang and Professor Schwager for their instruction and encouragement as well as their participation on my thesis defense committee.

To General Electric and the incredible educators there, thank you for all of your support throughout the Edison Engineering Development Program including the two wonderful years in A and B course in addition to this Masters. Your support has provided so much to my development and career.

In addition I would like to acknowledge the support of the IML by the Office of Naval Research through MURI Grant Number N00014-10-1-0952.

To my friends in the Intelligent Mechatronics Lab: Kayhan, Shuai, Zhaodan, Xi and Bowen. Thanks for making tough lab days enjoyable.

Steve, J., Brian, Tim, Ryan, Kyle and Allie. Thanks for your support during my

best and worst days. Without your perpetual heckling and support, I may have never begun or completed my thesis, let alone enjoyed myself while doing it.

Finally I'd like to thank my family and in particular my mother and father. You have supported me my entire life. Thanks for always listening to and believing in me, even when I didn't.

MOTION CONTROL USING OPTICAL FLOW OF SPARSE IMAGE FEATURES

J. PAUL SEEBACHER

ABSTRACT

Reactive motion planning and local navigation of robots remains a significant challenge in the motion control of robotic vehicles. This thesis presents new results on vision guided navigation using optical flow. By detecting key image features, calculating optical flow and leveraging time-to-transit (tau) as a feedback signal, control architectures can steer a vehicle so as to avoid obstacles while simultaneously using them as navigation beacons. Averaging and balancing tau over multiple image features successfully guides a vehicle along a corridor while avoiding looming objects in the periphery. In addition, the averaging strategy deemphasizes noise associated with rotationally induced flow fields, mitigating risks of positive feedback akin to the Larsen effect. A recently developed, biologically inspired, binary-key point description algorithm, FReaK, offers process speed-ups that make vision-based feedback signals achievable. A Parrot ARDrone2 has proven to be a reliable platform for testing the architecture and has demonstrated the control law's effectiveness in using time-to-transit calculations for real-time navigation.

Contents

List of Tables

List of Figures

List of Symbols and Abbreviations

A	.....	Varying matrices
b	.....	Varying vectors
$\vec{b}$	.....	Vector mask of filtered points
BRIEF	.....	Binary Robust Independent Elementary Features
BRISK	.....	Binary Robust Invariant Scalable Keypoints
C	.....	C-programming language
CPU	.....	Central Processing Unit
$curr$	.....	All current image information
$d_i(t)$	.....	object distance on image plane from the FOE
$\dot{d}_i(t)$	.....	First time derivative of $d_i(t)$ (optical flow)
$d(t)$	.....	Real world object distance from axis of motion
d_H	.....	Hamming distance between matched keypoints
DOF	.....	Degrees of Freedom
f	.....	Camera focal depth
F	.....	FReaK binary descriptor
FAST	.....	Features from Accelerated Segment Test
FFmpeg	.....	Open source multimedia handler
FIFO	.....	First in first out
FOE	.....	Focus of Expansion
FOV	.....	Field of view
FORTRAN	.....	Fortran programming language
FReaK	.....	Fast Retina Keypoint
GB	.....	Gigabyte
GIL	.....	Global Interpreter Lock
h	.....	Image height
h_c	.....	Image vertical center
Hz	.....	Hertz
$I(x, y, t)$	.....	Image intensity in space (x, y) and time (t)
IMU	.....	Inertial Measurement Unit
IO	.....	Input-Output
i, j, k	.....	subscript indices attached indiscriminately
kp_t	.....	keypoint from training image
kp_q	.....	keypoint from query image

LK	.....	Lucas and Kanade (authors of OF algorithm)
$\vec{m}_{cp}$	.....	Vector of keypoint matches
m, n	.....	number of rows and number of columns respectively
M, N	.....	Constants expressing a number of an item
MB	.....	Megabyte
NumPy	.....	Numerical Python
OpenCV	.....	Open Computer Vision library
p_i	.....	Pixel under test
$\vec{p}$	.....	vector filled with keypoints
PDF	.....	Probability Density Function
$prev$	.....	All previous (last iteration) image information
$\mathbb{R}$	.....	Domain of Real numbers
ROC	.....	Receiver operating characteristic
ROS	.....	Robot Operating System
SDK	.....	Software Development Kit
sgn	.....	Sign function. Returns sign of a number.
SIFT	.....	Scale Invariant Feature Transform
t	.....	time
τ	.....	Time-to-transit
τ_L	.....	Left panel time-to-transit
τ_R	.....	Right panel time-to-transit
TTT	.....	Time to Transit
UI	.....	User Interface
$v(t)$	.....	Observer velocity
v, v_x, v_y	.....	Image optical flow velocities
w	.....	Image width
w_c	.....	Image horizontal center
$x(t)$	.....	Distance between object and camera focal point
XOR	.....	Exclusive Or
(x, y)	.....	coordinate pair on the image plane

Terminology

Detector

Algorithm which determines notable keypoints within an image.

Descriptor

Keypoint relational taxonomy; an algorithm which produces such identifiers.

Focus of Expansion

Image equivalent to vanishing point in 1-point perspective. Point on image plane pierced by vehicle direction of motion.

Image Plane

Coordinate plane applied to an image. See figure 3·2.

Keypoint

Notable feature point in image plane having pixel coordinates (x_i, y_i).

Matcher

Algorithm to find matches between keypoints from each keypoint's descriptor.

Optical Flow

Flow vector describing the motion of a pixel between two frames of a video.

Time-to-Transit

Time before an object in camera field of view passes through the image plane.

Quadcopter

Aerial vertical take off and landing vehicle powered by four propellers.

Chapter 1

Introduction

1.1 Motivation

Evolution over hundreds of millions of years has produced animal vision systems that offer significant advantages for survival. Organisms such as bats, birds, fish, and insects rely on vision-based sensing for feature tracking and motion control. Taking inspiration from animal vision, many of the new artificial vision technologies that have been realized have very desirable characteristics such as low cost, high resolution, and large field of view. Furthermore, camera based systems have significant growth potential from primitive flow balancing strategies to highly aspirational feature recognition or object tracking. This thesis reviews control schema which use optical sensing to solve reactive navigation problems. Testing of the control laws was performed after developing image processing software and implementing it on a widely used air vehicle—the Parrot AR.Drone2.

Research has focused on implementing vision based control strategies that utilize relative spatial flow for robotic steering. Time-to-transit, the time before an image feature crosses the image plane, may be leveraged to avoid looming objects. For any control algorithm that might be implemented, work is required to determine how robust the feedback signal is against processing time, motion induced noise, and environmental influences like feature poor regions or in-field object motion.

Current state of the art systems rely on proximity-based sensors to determine distance and relative location of nearby objects. This approach requires numerous field

of view and/or range finding sensors. Because these systems rely on unidirectional sensors for recognition of nearby objects and obstacles, the probability of overlooking or misrepresenting nearby features is high and implementing robust systems is difficult.

A significant obstacle to an on-board vision-based motion planning system is its high level of computation complexity and resource requirements. This thesis demonstrates that coupling well-known vision algorithms—FAST for detection, FReaK for description and a brute force matching strategy—produces a reliable number of robust keypoints at rates accommodating feedback control. Lucas-Kanade sparse optical flow calculations further accommodate processing speed limits while demonstrating robustness to noise.

1.2 Background

Optical flow and visual servoing are pervasive throughout control literature and nature. Research performed in the Boston University Intelligent Mechatronics Lab adapts the optical flow signal into a generalized control strategies which derives the parameter, time-to-transit, from distinct yet sparse image features, [1], [2], [3], [4]. These control laws derive from observations of the bat species *Myotis velifer*, and particularly how the bats use distant objects as navigation beacons, employing time-to-transit as the feedback signal. These works define, reiterate and introduce key terms such as time-to-transit, the time before an image feature crosses the image plane; loom, the inverse of time-to-transit and an indicator of object closeness; and salience, notable and distinct features which may be tracked on the image plane. Other works introducing or evaluating concepts like time-to-contact, time-to-passage and loom include examinations of diving gannets, collision avoidance in pigeons, or cars, and evaluation of heading direction [5], [6], [7], [8].

Similar control systems have been developed which rely on image signals and attempt to balance optical flow information to navigate vehicles through canyons, or hallways using two laterally facing cameras [9], [10]. Optical flow is used as a raw input to control vehicle forward motion rate using a lateral camera while avoiding collisions determined by a forward facing camera [11]. Other systems use optical flow to estimate velocity and determine vehicle pose [12].

All of the above systems vary with regard to optical flow calculations, keypoint introduction, and keypoint tracking. These methods range from dense to sparse optical flow methods [13], [14], and make use of a variety of feature detectors and descriptors as well, including FAST [15], [16], SIFT [17], and BRIEF [18].

1.3 Statement of Work

Significant speed-ups in optical flow computation, keypoint tracking and control calculation are developed by this thesis, making real time visual feedback systems viable on board single camera robotic vehicles. The system ties together rapid and robust point introduction by coupling FAST detection with FReaK description. Keypoints are filtered before being introduced using a Lowe ratio test. Points are tracked through frame histories using FIFO styled data structures which quickly augment or trim entering and exiting points. Optical flow calculations are performed using Lucas-Kanade sparse optical flow optimizations in forward and reverse to eliminat drifting points. Time-to-transit is calculated for all persistent points and the values are averaged using a 2-dimensional histogram strategy. Finally, a balancing scheme is employed to test vehicle motion direction, threat direction and style of motion in order to produce a viable control signal which avoids looming obstacles.

The software implementation was deployed onto a quadcopter vehicle, a Parrot AR.Drone2. This was flight tested in order to evaluate the software capability. The

results indicate that the deployed strategy is effective at balancing vehicle motion along an obstacled corridor.

The primary deliverables offered by this project are:

- A package to perform all image processing, introduce and track keypoints, calculate optical flow and filter poorly defined, non-robust keypoints.
- Working control strategies which plug into the vehicle control system and use the image processing package to output control signals.
- A software interface with the Parrot AR.Drone2 that allows simple replacement of control components for rapid development.
- A software video interface for reading, writing and live demonstrations of the image processing library.
- A working, flying testbed, on which to examine control strategies, visualize controller logic and prepare for new enhancements.
- Documentation and explanation of work in the form of this thesis as well as code comments.

Chapter 2

Background

2.1 Optical Flow

Optical flow corresponds to the apparent velocity of a region of interest, expressing the region's movement, or its relative movement, between two images. The concept of optical flow has pervaded the computer vision literature since its introduction in the late 1970's and early 1980's in seminal works such as *Determining Optical Flow* written by Horn and Schunk [13].

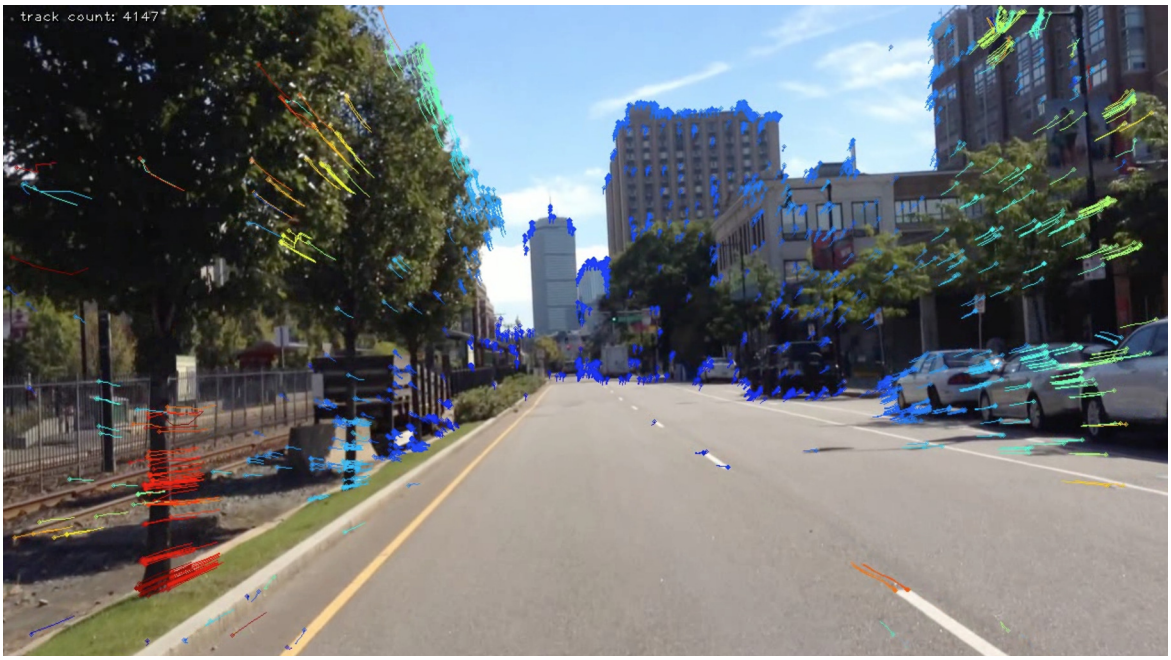


Figure 2·1: Sparse optical flow field for forward moving camera. Note how flow diverges where the road meets the horizon (focus of expansion)

Optical flow relates to real world motions through the following equation

$$\phi_{flow} = -\omega(t) + \frac{v(t)}{r(t)} sin(\theta(t))$$

where $v(t)$ represents object or observer velocity, $\omega(t)$ object or observer rotational velocity, $r(t)$ distance between object and observer, and $\theta(t)$, the angle between the direction of motion and the image plane.

Optical flow may be perceived when considering a scene viewed from a moving vehicle as illustrated in figure 2·1. As one travels along facing forward, objects viewed far down the road appear to move slowly, whereas objects viewed in the periphery seem to rapidly transit through the edge of the field of vision. Looking down a straight road with no visual obstructions, the point where the road meets the horizon is known as the focus of expansion. Plotting optical flow for a motion that follows the road demonstrates how each flow vector exists on a ray emanating from the focus of expansion.

Figure 2·2: Sparse optical flow field for sideways facing camera. Note how flow is much greater for objects closer to the camera

Alternatively, consider a new vantage point from the moving vehicle, looking from a side window, 90° relative to the direction of travel. Notably, distant objects move slowly across the field of vision while nearby objects pass by rapidly, almost a blur.

In the previous two examples, the objects in the scene remained stationary while the observer moved along in a vehicle. The objects moving in the field of vision are said to have *relative flow* or *relative motion* within the image plane, with flow magnitude scaled by observer velocity and object distance. The optical flow generated by the motion of an observer is critical to work presented in this thesis and is the basis for all proposed control strategies.

Still, it is worth noting an alternate scenario in which the observer is stationary while the objects move within frame. Take for instance, a view of a busy highway. The calculation for optical flow is the same in either instance, however, these two edge cases: moving observer - stationary scene and stationary observer - moving scene, enable certain simplifications to determine object distance or absolute speed.

A most complex case would be to consider a moving observer tracking a moving object. In this case, both the flow of the observer and the flow of the object result in a calculated optical flow, which add as one might expect, through superposition. However, information about absolute velocities of objects becomes muddled as optical flow is scaled by both the velocity of the object and of the observer. This scenario is outside the scope of this work.

As presented, optical flow is valuable for sensing relative flow rates within the frame. In order to calculate the velocity of an object or the distance between the observer and an object, additional information is needed to scale the flow vectors, for instance, if the object size is known a priori. Despite the difficulty sensing absolute velocities, a way forward may be derived from the optical flow signal, in the form of a parameter that can be used as a viable control signal. This parameter is known as

Figure 2·3: Vehicles moving on a highway. Optical flow for stationary observer viewing moving objects.

time-to-transit.

2.2 Time to Transit

Time-to-transit indicates the time before a moving observer (or object) travels beyond the field of view of an object (or observer).The term stems from a number of papers in the field of behavioral neuroscience; scientists coined *time-to-collision* after recognizing its relevance when considering a number of animal driven avoidance and tracking behaviors such as human vehicle braking [7], diving gannets [5], and stimuli in the brains of pigeons [6]. Time-to-contact refers simply to the span of time before an object passes through the image plane. Since then, time-to-contact and time-to-transit have gained popularity in controls engineering, for both side-facing cameras [1], [9], or forward facing cameras [3], [19]. Time-to-transit may be calculated using the distance between an object and the focus of expansion, $d_i(t)$, and the

Figure 2·4: Optical flow for moving observer with moving object. A jet fighter banks away from the cockpit of a travelling fighter. Vehicle motion is evident when observing flow field on ocean. Because fighters travel at the same speed, the high rate of speed of each fighter is not captured.

associated optical flow component along $d_i(t)$, $\dot{d}_i(t)$, as follows:

$$\tau = \frac{d_i(t)}{\dot{d}_i(t)} \tag{2.1}$$

Dividing the image plane distance $d_i(t)$ by its optical flow components $\dot{d}_i(t)$ eliminates the scale uncertainty for feature sizes in the image. The resulting value is the time remaining before an object passes through the image plane. More intuitively, time-to-transit may be viewed as a similar triangles problem:

As seen in figure 2·5, similar triangles may be constructed between the image plane and 3D scene through the focal point of the camera. Here, d(t) represents real world distance of an object from the vehicle axis of motion while x(t) represents the component of real world distance between the object and the camera focal point along

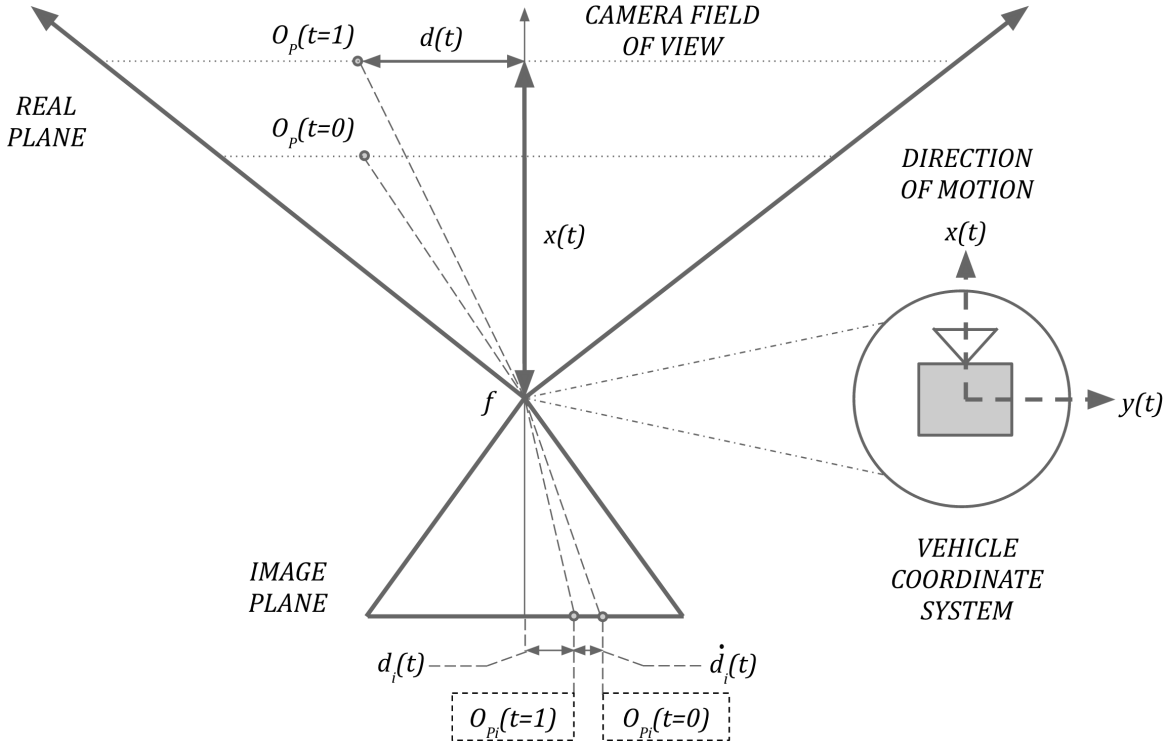


Figure 2·5: Time-to-transit derivation is evident geometrically using similar triangles.

the axis of motion.

$$\frac{d(t)}{x(t)} = \frac{d_i(t)}{f} \tag{2.2}$$

Objects in the image plane scale with the camera's intrinsic focal length, f, however absolute sizes are indeterminate without object size or distance from the camera. As the camera or the object begins to move with constant velocity v, the relative motion and rate of motion may be leveraged to form a differential equation. Rearranging and differentiating equation 2.2 yields:

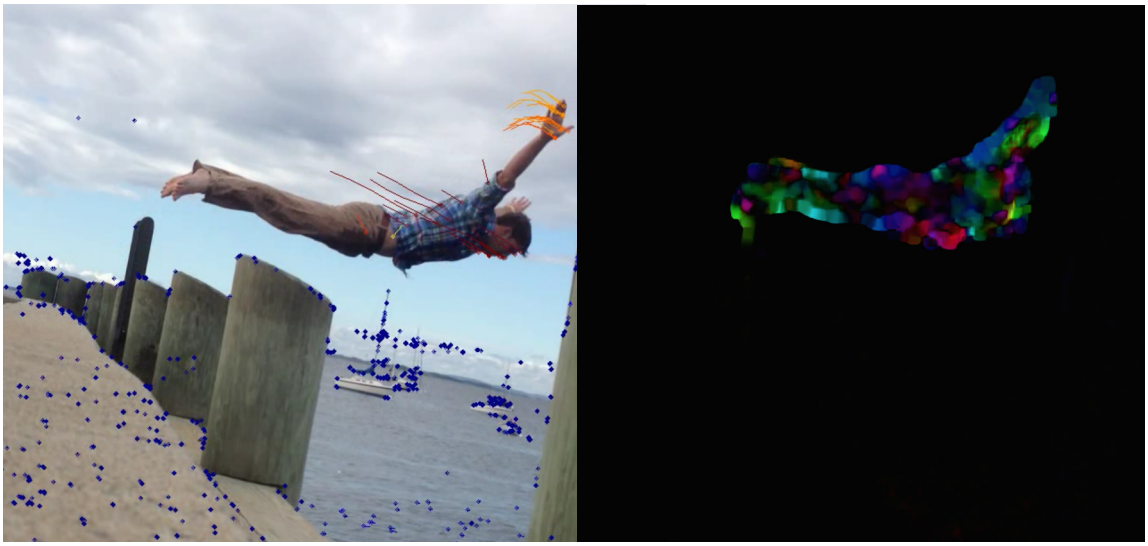
$$\dot{d_i}(t) \cdot x(t) - v \cdot d_i(t) = 0 \tag{2.3}$$

which may be further adapted into the particularly useful form:

$$\tau = \frac{d_i(t)}{\dot{d}_i(t)} = \frac{x(t)}{v} \left| \begin{array}{ll} d_i(t) & \in [0, \infty) \\ \dot{d}_i(t) & \in \mathbb{R} \\ x(t) & \in [0, \infty) \\ v & \in \mathbb{R} \end{array} \right. \tag{2.4}$$

Equation 2.4 demonstrates how τ captures real world distance and velocity information, but most importantly, that τ may be calculated using information obtained entirely from the image plane. Therefore, observing and linking τ to real world robot and object behaviors provide guidelines for using τ as a control signal. Consider the domain of values τ may assume. Again observing equation 2.4, it is clear that $\tau \in \mathbb{R}$. This range may be broken up into a number of notable cases which are reviewed below.

- Images where $\forall \tau \in (-\infty, 0)$, indicate the vehicle or object is retreating.
- Images where $\forall \tau \in (0, \infty)$, indicate the vehicle or object is advancing.
- Images where $\exists \tau \in (-\infty, 0)$ and $\exists \tau \in (0, \infty)$, indicate the vehicle has motion components which are highly skewed from the normal axis of the image plane.
- Features with $\tau = 0$, indicate features which are passing through the image plane. Additionally, in some instances, features transiting within ϵ pixels of the FOE may also have $\tau \sim 0$
- Features where the distance vector projection of optical flow, $\dot{d}_i(t)$ is near zero, result in infinite values despite distance. These objects may or may not be approaching the vehicle.



(a) (b)

Figure 2·6: (a) Sparse optical flow field (Lucas-Kanade); and (b) Dense optical flow field (Farneback).

2.3 Sparse Optical Flow Methods

A number of algorithms exist that perform optical flow calculations. As mentioned previously, one popular implementation was derived and published by Berthold K.P. Horn and Brian G. Schunck [13] in 1981. This method is a global method, meaning optical flow is calculated densely for each pixel in the image in order to minimize a global constraint. An alternate, highly popular method produced by Bruce D. Lucas and Takeo Kanade, uses local, predetermined feature points to calculate sparse optical flow by optimizing an energy function at each given point [14]. Local methods offer high robustness in the presence of noise but do not offer complete information about the flow field (sparse flow). Alternatively, global methods offer dense flow fields, reporting information throughout the entire image, but are known to be more susceptible to noise [20]. Finally, local methods require fewer calculations which results in faster processing speeds. The processing improvements and robustness to noise offered by local methods align with the goal of this thesis and hence, Lucas-

Kanade was selected for optical flow calculations.

After a first order taylor expansion, the optical flow equation may be written in its most popular form as:

$$\frac{dI}{dx}v_x + \frac{dI}{dy}v_y + \frac{dI}{dt} = 0 \tag{2.5}$$

where v_x and v_y are the x and y component optical flow velocities and $\frac{dI}{dx}$, $\frac{dI}{dy}$ and $\frac{dI}{dt}$ are the rates of change of image intensity in the spatial, x and y, and temporal, t, dimensions respectively.

Lucas and Kanade suggested that the motion between two adjacent frames is small and approximately constant for pixels within a neighborhood. Using this assumption, each pixel p_i in a predefined region, must satisfy the equations:

$$\frac{dI}{dx}(p_i)v_x + \frac{dI}{dy}(p_i)v_y + \frac{dI}{dt}(p_i) = 0 \mid i = 1 \dots n \tag{2.6}$$

which leads to an over defined system in two unknowns. By applying a least squares optimization, v_x and v_y may be determined for the region by solving the system of equations with:

$$A = \begin{bmatrix} \frac{dI}{dx}(p_1) & \frac{dI}{dy}(p_1) \\ \vdots & \vdots \\ \frac{dI}{dx}(p_n) & \frac{dI}{dy}(p_n) \end{bmatrix} \quad v = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \quad b = \begin{bmatrix} \frac{dI}{dt}(p_1) \\ \vdots \\ \frac{dI}{dt}(p_n) \end{bmatrix} \tag{2.7}$$

$$\begin{aligned} A^T Av &= A^T b \\ v &= (A^T A)^{-1} A^T b \end{aligned} \tag{2.8}$$

Lastly, to perform sparse optical flow calculations, a class of algorithms is needed to detect robust key points which are consistent and track-able between video frames. These feature points define the region on which the LK-optical flow algorithm may be applied. Such classes of algorithms are known as feature detectors, descriptors and matchers.

2.4 Feature Detectors, Descriptors and Matchers

Determining robust feature points to pass into the Lucas-Kanade optical flow algorithm requires three stages: feature detection, feature description, and feature matching. These algorithms are applied in order to introduce new, robust, feature points, as well as retain previously introduced points of interest. The features should be invariant to pose changes i.e. rotation, skew or location; be distinctive, with high detection rate and low false positive rate; and be repeatable, meaning the same points should be detectable despite changes in viewing conditions.

Ultimately, the algorithm must determine and retain consistent object features while the object persists in the video. Such a strategy ensures that any object entering the frame may be represented and utilized to avoid collisions, track objects of note or guide the vehicle. A number of algorithms are suitable for this purpose but, for the sake of brevity, this paper only details the algorithms which are ultimately implemented. Still, a brief review detailing the selection process for these algorithms is provided in *chapter 4 Experimental Results*.

2.4.1 Feature Detectors

Feature detection algorithms are responsible for determining notable points within an image. The primary groups of notable features are edges (areas with large image gradient), corners (areas with large image gradient and high curvature) or blobs (areas with complimentary image structure). Typically, detectors are designed to recognize only one type of feature. Detectors are graded based on repeatability, which is their ability to detect consistent features over multiple views of the same scene. Computational performance and speed are two other desirable traits.

To apply control laws using feedback from image parameters requires near real-time image processing. The most suitable detector for real time applications is the

Features from Accelerated Segment Tests, or FAST detector.

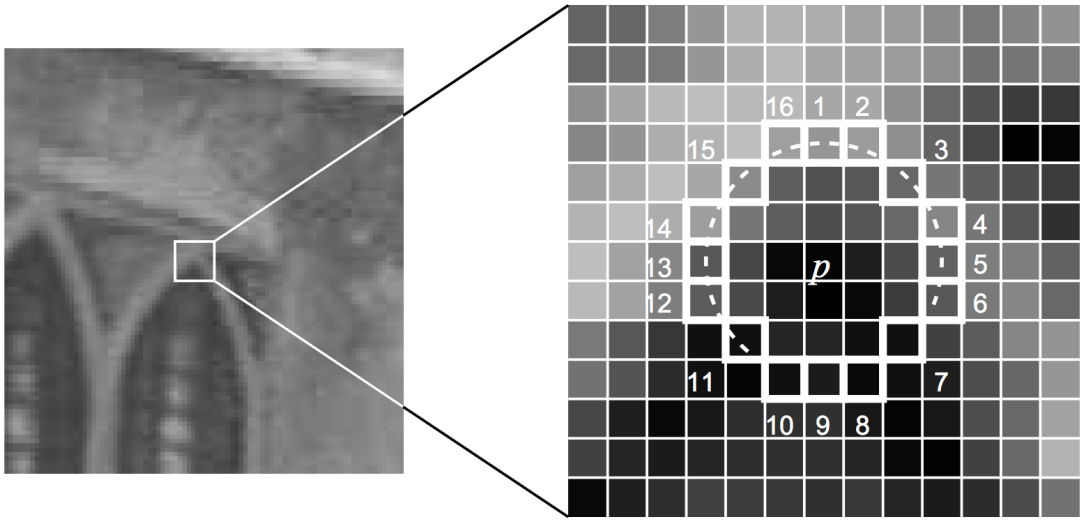


Figure 2·7: FAST: Corner detection from segment tests. [15]

Created by Edward Rosten and Tom Drummond, the algorithm detects corner points by examining pixels of a Bresenham circle. If N contiguous pixels exist which are brighter (or darker) than the candidate pixel, p, at the center of the circle, the point p is marked as a corner and its position is returned. An example with $r = 3$ pixels is demonstrated in figure 2·7.

Furthermore, for $N \geq 12$, a rapid rejection technique is employed to reduce computation time. The algorithm tests if at least three of four pixels (1, 5, 9 and 13 in the image above) have differing intensity than the pixel under test. If so, the algorithm continues to test intermediary points, but, because corners are the exception rather than the rule, the rejection technique results in a massive reduction of comparisons, down to an average of 3.8 per point [16]. For this reason, FAST is considered the most rapid feature detection algorithm available.

With key features defined in an image, the algorithm must now determine the features most likely to persist and be matched in following frames. For this, a method is required that both describes a feature and also allows for fuzzy comparisons of two

features.

2.4.2 Feature Descriptors

Comparing keypoints between images requires a robust method for *defining* those keypoints. Feature descriptor algorithms solve this problem by using local information around the point to uniquely define that feature. Besides the unique definition, descriptors may offer orientation information as well as properties which make them scale or rotation invariant. While desirable, these characteristics may also be imparted at the detection level. Descriptor algorithms typically consume the largest computation time; prescribing unique classifiers for points requires expensive computations over a large neighborhood around the key point.

Over the past decade, the pervasiveness of mobile phones and the resulting image revolution have driven rapid improvements in description algorithms. The baseline descriptor, SIFT, introduced by David Lowe in his seminal 2004 work, *Distinctive Image Features from Scale-Invariant Keypoints*, is highly regarded for its simplicity and invariance to scale and rotation [17]. Additionally, its performance and accessibility have led to its persistence and prevalence over the past decade.

Figure 2·8 demonstrates how the SIFT descriptor is created by compiling information obtained from 2D histograms of pixel intensity over members of a larger pattern i.e., boxes of a grid. The information extracted from the histograms describe both orientation and composition of individual regions. The regional information is compiled to produce the descriptor string. Descriptors may then be compared based on similarity, orientation and euclidean distance.

One notable improvement to description methods came with the introduction of binary string descriptors as proposed in *BRIEF: Binary Robust Independent Elementary Features* [18]. Rather than define regions with information obtained from binned histogram values, the descriptor builds a binary string by comparing image intensities

over many pairs of points selected from different regions within a sampling pattern. The simple comparison of image intensity reduces computational time and complexity. Additionally, comparing descriptors between images is simplified, where hamming distance (which yields Manhattan distance when comparing binary strings) may be used in lieu of an ℓ^2-norm by performing a bitwise XOR between two descriptors.

The newest binary descriptor, FReaK, an acronym for Fast REtinA Keypoint, hones previous binary description techniques and introduces a number of optimizations derived from observations of the human eye [21]. The circular sampling pattern used by FReaK is inspired by ganglion cell density in the human eye. To retain detailed feature information, the eye has a high density of ganglion cells responsible for observing the region of focus. Cell density decays exponentially in the neighborhood surrounding the region of focus but, the less dense cells are important for determining

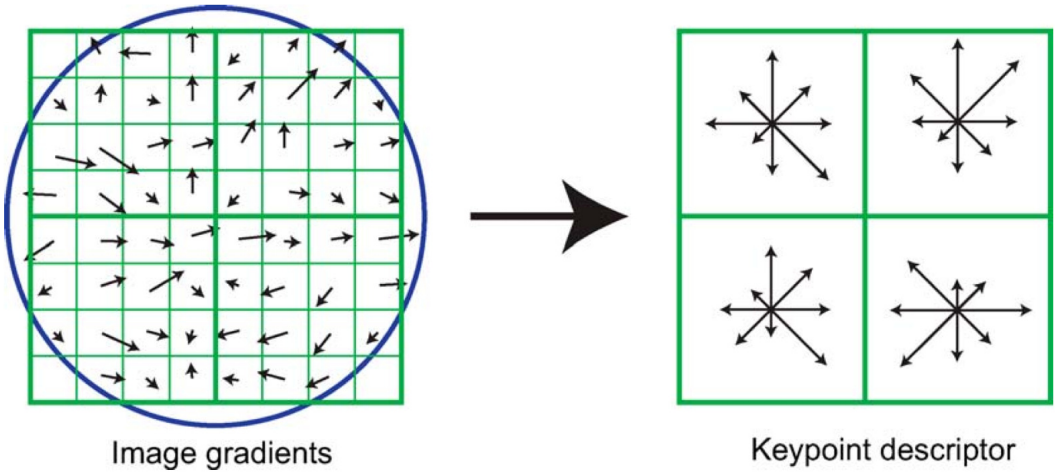
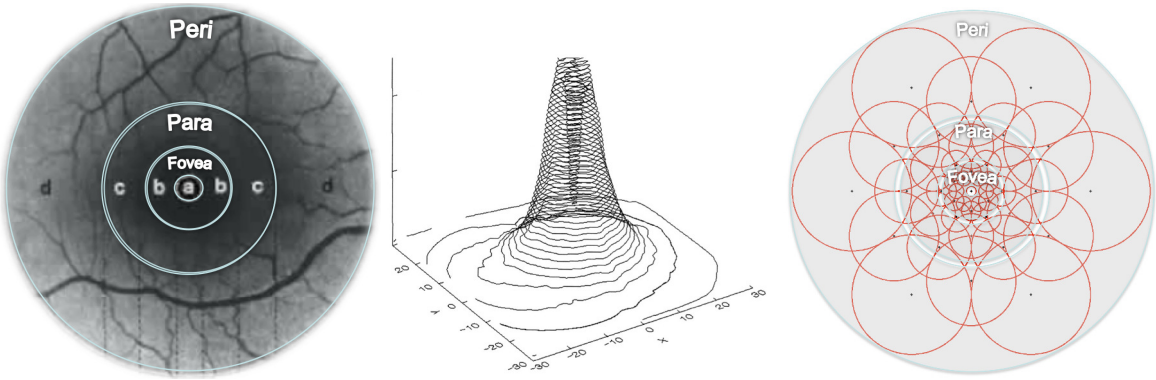


Figure 2·8: "A keypoint descriptor is created by first computing the gradient magnitude and orientation at each image sample point in a region around the keypoint location, as shown on the left. These are weighted by a Gaussian window, indicated by the overlaid circle. These samples are then accumulated into orientation histograms summarizing the contents over 4 x 4 subregions, as shown on the right, with the length of each arrow corresponding to the sum of the gradient magnitudes near that direction within the region.". Quote and image credit to [17]



(a) Areas of Retina [22] (b) Ganglion cell density over retina [23] (c) FReaK Sampling pattern [21]

Figure 2·9: FReaK sampling pattern shape and Gaussian kernel σ-value are scaled and adapted from ganglion cell density in the human eye

orientation information.

The FReaK descriptor is built similarly to the human retina. The sampling pattern, demonstrated in figure 2·9c, is composed of a number of receptor fields—individual circles in the pattern—on which a difference of Gaussians is performed between the image and a Gaussian smoothing kernel. The radius of each circle represents the standard deviation of the Gaussian smoothing kernel applied to that region. The density of receptor fields is greatest near the keypoint being described. Sampling density decreases exponentially in the neighborhood of the keypoint but is essential for defining descriptor orientation. Additionally, the circular sampling patterns overlap, introducing smoothing between adjacent sample regions. The authors of the algorithm observed that by overlapping receptor fields, the number of receptor pairs necessary to create unique description strings was reduced. Higher efficiency receptor pairs ultimately yield a smaller sampling pattern, improving calculation speeds.

FReaK descriptors are binary strings constructed using the following equation,

$$F = \sum_{0 \leq a < N} 2^a T(P_a) \tag{2.9}$$

where P_a is a pair of receptor fields and N is the length of the descriptor binary string, and

$$T(P_a) = \begin{cases} 1 & \text{if } |I(P_a^{r_1}) - I(P_a^{r_2})| > \epsilon, \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

where $I(P_a^{r_1})$ is the smoothed intensity from the Gaussian kernel applied to the first receptive field of the pair P_a [21].

In practice, the algorithm uses a learning phase to select the best 512 receptor field pairings that will be used to create the descriptor. Recalling that each paring results in a binary 1 or 0, the primary component of the descriptor (without orientation components) is 64 bytes. Using the forty-three receptor fields as demonstrated in figure2·9c, exactly 903 comparison pairs exist for each descriptor in the image.

To select the best 512 pairs, consider a matrix with n-descriptors (rows) and $m = 903$ receptor pairings (columns). The column-wise averages over all descriptors in the image is computed and the 512 columns with highest variance (those with mean closet to 0.5 for a binary average) are selected. This learning phase maximizes potential for uniquely defined descriptors and is only performed once per comparison on the training image (the first image being compared).

Orientation is determined by applying a distance scaled weighted average to the result of the difference of Gaussians between two paired regions.

Mathematically,

$$O = \frac{1}{M} \sum_{P_o \in G} (I(P_o^{r_1}) - I(P_o^{r_2})) \frac{P_o^{r_1} - P_o^{r_2}}{\|P_o^{r_1} - P_o^{r_2}\|} \tag{2.11}$$

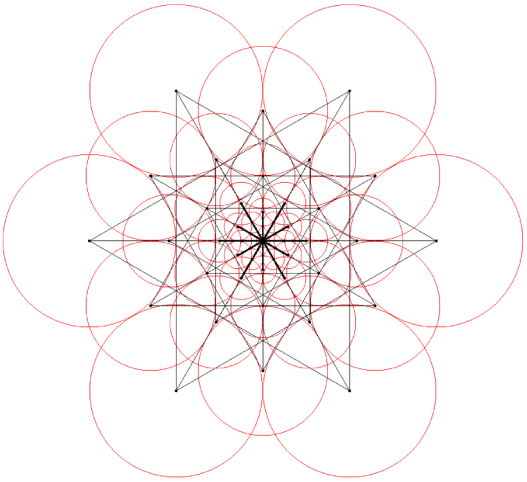


Figure 2·10: Predefined pairs of sampling region used for calculating FReaK descriptor orientation [21].

Where M is the number of pairs in G and $P_o^{r_i}$ is the 2D vector of spatial coordinates originating from the center of the sampling pattern [21]. The sampling pattern and pairings are predefined for the orientation estimate as shown in figure 2·10. In tests, the FReaK descriptor is 50% faster to compute than the next closest algorithm and requires about $1/5^{th}$ the memory. These two facets make it highly suited for real time applications. The final stage of keypoint selection is matching.

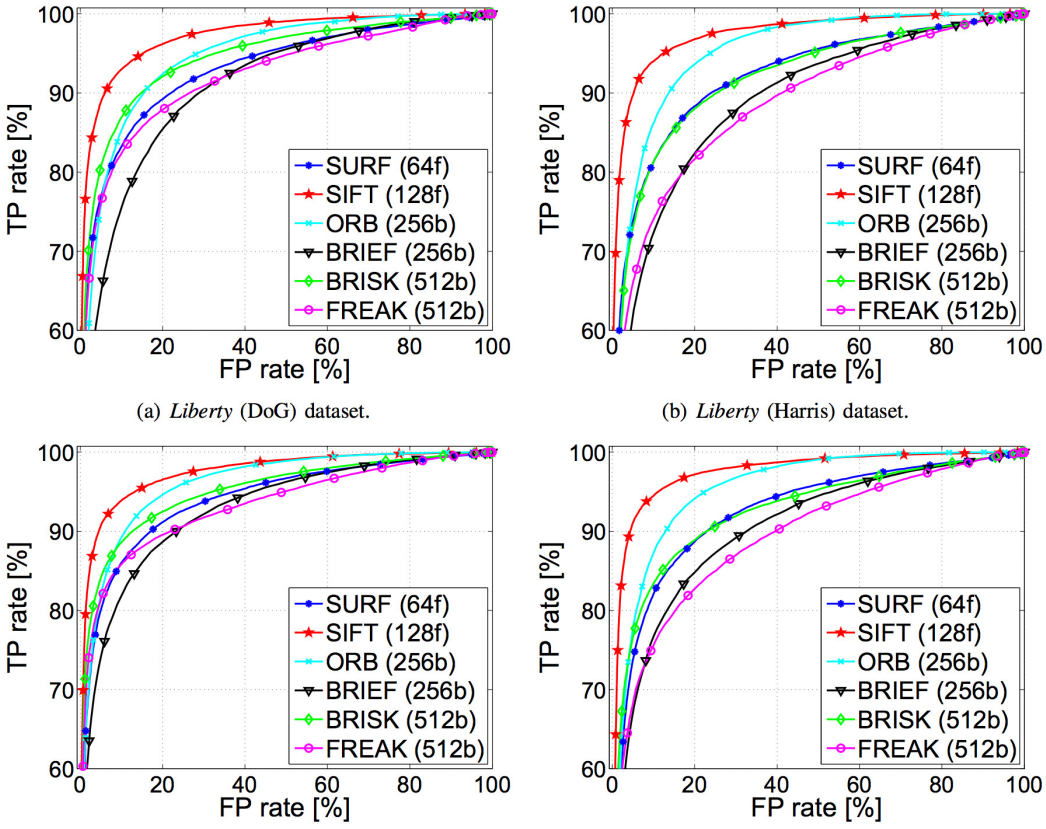
2.4.3 Feature Matchers

Feature matching falls into two categories; library matching and cross image matching. Library matching considers keypoints from an image database, however it is outside the scope of this work [24]. Cross image matching considers how descriptors may be matched between two images viewing the same scene.

The standard method is a straightforward brute-force technique. A single descriptor from the training image is compared to each descriptor in the test image. For binary detectors, rejection is designed to occur early in the descriptor. For FReaK, over 90% of possible matches are rejected in comparisons of the first 16 bytes of the

descriptor [21]. Lastly, matches are returned with, and sorted by, their distance from the original keypoint. Further filtering and rejection may be performed based on descriptor distances or back matching from the test image to the training image.

The generally adopted evaluation metric for matchers and descriptors is presented by Mikolajczyk and Schmid [25]. One intuitive metric is a plot based on the number of true positives (correct matches) divided by the total possible correct matches versus the number of false positives (incorrect matches) divided by the total number of actual rejections. Plotting these terms against one another produces a receiver operating characteristic (ROC) curve. The best matchers seek to maximize the area under this curve. The results of a recent paper which evaluates matching for a number of descriptors are summarized in figure 2·11.



(a) *Liberty* (DoG) dataset.

(b) *Liberty* (Harris) dataset.

(c) *Notredame* (DoG) dataset.

(d) *Notredame* (Harris) dataset.

Figure 2·11: ROC evaluation of Bruteforce-matching applied to various descriptors over four datasets [26]. The numeric value following the descriptor name–i.e. 64f and 512b–represents the number and type of data used to store the descriptor–64 floats (4 bytes each) and 512 bits respectively.

Chapter 3

Implementation

To test the capability of image based control schemes, a software platform was developed using the algorithms described throughout *chapter 2*. The platform was implemented entirely in the Python programing language which was chosen for its massive collection of libraries, rapid prototyping capability, readability, and popularity. The software was designed modularly, so that it might be deployed in pieces, as needed, onto many different hardware environments. Additionally, it was designed assuming applications demanding real time vision based control. For this reason, achieving high processing speeds stood as the principal requirement for the project.

Some might consider this tenet at odds with the selected programing language, however, the contradiction is mitigated when considering ported libraries. In other words, each computationally demanding task, i.e. image processing, calculations, video handling, etc. is performed by highly optimized and pre-compiled C and Fortran code ported into python. The two libraries most heavily leveraged are OpenCV [27] for computer vision processing and NumPy [28] for numerical computations.

The software is separated into distinct modules which are discussed individually: image processing, control and AR.Drone2 operating system. The general flow diagram describing the image processing system is demonstrated in figure 3·1. A section is also set aside to review multi-threading which is prevalent throughout the merged platform. Discussion of the video handling routines is performed in the comments for the video processing code demonstrated in the appendix.

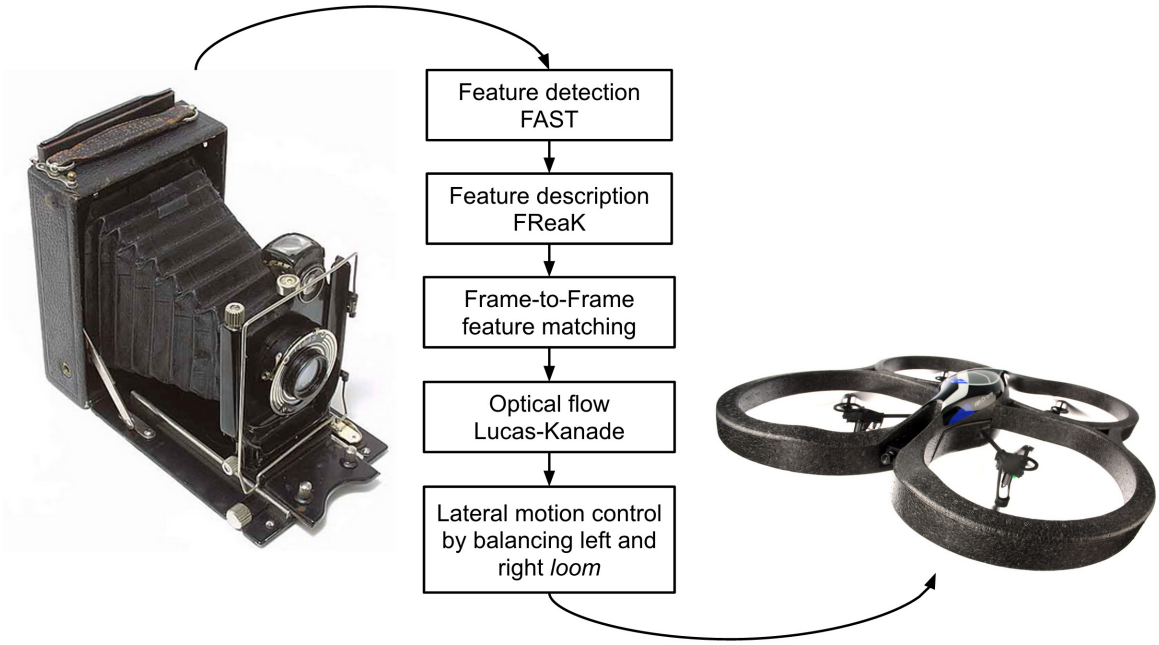


Figure 3·1: Software and control, general overview.

3.1 Image Processing

Two classes of objects were created to encapsulate all necessary components for video image processing.

- `Image` is a container class responsible for storing individual frames of a video in color and gray scale. It retains inherent information regarding image size and associated 'capture' time. After image processing is performed, the object stores notable keypoints, their associated descriptors, and a historic record (tracks) of keypoint positions which have persisted through recent frames. The keypoint, descriptor and track data is set lazily, whereas, the other properties are set during instantiation.

- `ImageCompare` is a class, designed for image processing between pairs of images. It receives `Image` objects holding frames from a video feed and updates each `Image` with keypoint, descriptor and track information. The object han-

dles construction and destruction of OpenCV detector, descriptor and matcher objects, maintains time records, introduces keypoints efficiently, applies filtering techniques to select persistent keypoints, tracks keypoint motion, performs optical flow computations, and offers a range of visualization functions to the user. The details describing these actions are explained in the following subsections. Again, the class was designed around the core tenet of real time video processing. System performance is presented in chapter *4 Experimental Results*.

An overview of the image processing architecture may be considered in algorithm 1. Each time a video frame is passed into the compare method of the `ImageCompare`

Algorithm 1 State Machine Updates

function COMPARE(Image) ▷ Called once on successive video frames
  TRACK_TIME(Image)
  $prev \leftarrow curr$ ▷ Update states
  $curr \leftarrow$ Image
  $(\vec{p}_{new}, \vec{p}_{old}) \leftarrow$ OPTICAL_FLOW($curr$)
  return $(\vec{p}_{new}, \vec{p}_{old})$
procedure TRACK_TIME(Image) ▷ Maintains time for object
  $t_0 \leftarrow t_1$
  $t_1 \leftarrow$ Image.time
  assert $t_1 - t_0 > 0$
  $\Delta t \leftarrow t_1 - t_0$

class, internal data is advanced through a length two FIFO buffer. Data includes frame references, time, and feature points. Class storage is greatly reduced by employing the `Image` container class. The FIFO concept was refined after a number of software iterations, including one with a variable length buffer. Ultimately, elegance was chosen over power, which greatly simplified the top level code. The primary assumption of this handler structure is that frames are input sequentially in time. An error is raised if the Δt calculation is negative. This mitigates user end errors which might result in miscalculations of optical flow.

All comparative image processing is handled by the `optical_flow` method. There are two overall goals associated with the method: point introduction and point tracking. The general form of this code is summarized in algorithm 2.

Algorithm 2 Introducing and Maintaining Point Correspondences

function OPTICAL_FLOW($curr$, $prev$)
  $detect \leftarrow$ False
  if length($\vec{p}_0$) < threshold **then** ▷ Keypoint Introduction
    $detect \leftarrow$ True
    $curr \leftarrow$ FEATURE_DETECTION($curr$)
    $curr \leftarrow$ FEATURE_DESCRIPTION($curr$)
    $\vec{m}_{cp} \leftarrow$ MATCH($curr$, $prev$)
    $\vec{p}_{np} \leftarrow$ LOWE_FILTERING($\vec{m}_{cp}$)
    $A \leftarrow$ CREATE_TRACKS($\vec{p}_0$)
    $\vec{p}_0 \leftarrow$ concat($\vec{p}_{np}$, $\vec{p}_0$)
  if length($\vec{p}_0$) > 0 **then** ▷ Keypoint Tracking
    $\vec{p}_1 \leftarrow$ LK_OPTFLOW($prev$, $curr$, $\vec{p}_0$) ▷ forward LK-flow
    $\vec{p}_{0r} \leftarrow$ LK_OPTFLOW($curr$, $prev$, $\vec{p}_1$) ▷ backward LK-flow
    $\vec{b} \leftarrow$ FILTER_TRIM($\vec{p}_0$, $\vec{p}_{0r}$, $\vec{p}_1$)
    $(\vec{p}_{1f}, \vec{p}_{0f}) \leftarrow (\vec{p}_1 \wedge \vec{b}, \vec{p}_0 \wedge \vec{b})$
    $A \leftarrow$ UPDATE_TRACKS($\vec{p}_{1f}$, $\vec{b}$)
    if (length($\vec{p}_{1f}$) < threshold) & $\sim detect$ **then** ▷ Preliminary Introduction
      $curr \leftarrow$ FEATURE_DETECTION($curr$)
      $curr \leftarrow$ FEATURE_DESCRIPTION($curr$)
  $\vec{p}_0 \leftarrow \vec{p}_{1f}$
  return $(\vec{p}_{1f}, \vec{p}_{0f})$

In the **point introduction** component, *detect* is a boolean flag indicating whether feature description and detection have occured, $\vec{p}_0$ are keypoints detected within the previous frame, *curr* and *prev* are `Image` objects storing data for the current and previous frames, $\vec{m}_{cp}$ is a vector containing 2-tuples holding the two closest matches within the current image relative to a keypoint from the previous image, $\vec{p}_{np}$ is the vector of filtered and newly introduced (x,y) keypoint coordinates. The image coordinate system used by OpenCV is demonstrated in figure 3·2.

In the **point tracking** component, $\vec{p}_1$ is a vector of point's (x,y) coordinates in the query image (current image), after applying the LK-optical flow criteria to points $\vec{p}_0$ from the training image (previous image), $\vec{p}_{0r}$ are the back-computed points in the training image, after applying the LK-optical flow criteria to the recently calculated points, $\vec{p}_1$ from the query image, $\vec{b}$ is a boolean array equally sized to vector $\vec{p}_1$ which has True values for points meeting filtering requirements, $\vec{p}_{1f}$ and $\vec{p}_{0f}$ are the paired vectors of filtered points which match from the previous to current frame, and A is the tracking matrix, whose rows are unique keypoints and whose columns are the historic frame record of (x,y) values.

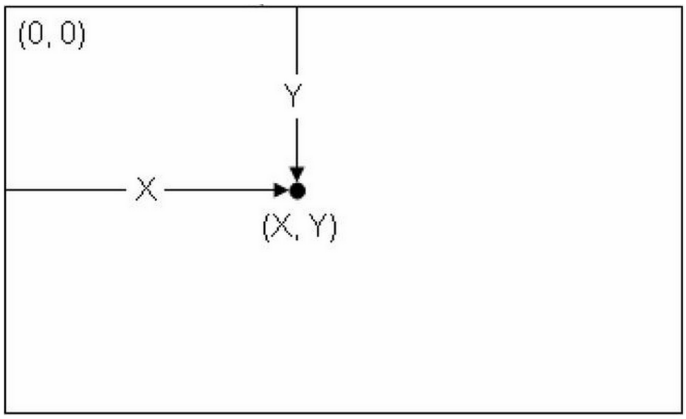


Figure 3·2: Image coordinate system in OpenCV

3.1.1 Point Introduction

Robust point introduction drives performance for feature tracking and control calculations. A moving observer requires additional performance enhancements; motion causes feature volatility and impermanence. To meet these demands, point introduction is performed constantly to replenish keypoints and maintain a predetermined threshold. Filtering increases the number of robust keypoints by limiting the number of falsely introduced matches and admitting only the strongest matches.

Observing algorithm 2, it is clear that points may be introduced for almost any

call to the `OPTICAL_FLOW` function. Only one exception exists, the first call, where a second image is not available for comparison. Point introduction is initiated when fewer keypoints exist than a globally defined threshold. This is possible when entering or exiting the function, where motion has caused keypoints to exit the frame or filtering has reduced the number of points below the threshold.

Feature detection is performed using OpenCV's grid adapted, FAST feature detector. The detector uses the algorithms detailed in *2.4.1 Feature Detectors* paired with a heuristic grid adaptation to select points uniformly over the partitioned image. The grid size, 6 x 6, is selected with the control balancing strategy in mind. Further explanation for this is presented in *3.2.2 Control Overview*. The detector is further constrained by a parameter which controls the total number of points to detect. This adjustment is critical for limiting superfluous downstream computations. The feature detector returns a vector of 2-tuples representing keypoint coordinate pairs, (x,y), from the image plane. Points are presorted based on feature response so that keypoints with strongest response are introduced first, improving overall point persistence.

Feature points detected by FAST are passed into FReaK description algorithm explained in *2.4.2 Feature Descriptors*. The descriptors are computed and lazily set to the associated image object in preparation for matching. Because the FAST detector is rotationally indiscriminate, the descriptor is implemented with orientation characterization enabled, improving rotation invariance. After keypoints and descriptors are defined for the current and previous frames, a brute force matching algorithm is executed. The method compares the descriptor from the training image against all points in the query image, quickly discarding non-matching points. Using the set of points from the query image whose descriptor matches the point in the train image, the method returns two points in closest proximity to the train point.

A pair of closest matches, rather than a single match, is needed to perform filtering before point introduction. For any given train keypoint, $kp_t(i)$, and the l-th proximal match from the query image, $kp_{q(\ell)}(i)$, where $l \in \{1, 2\}$, the hamming distances, $d_{H(\ell)}(i)$, between $kp_t(i)$ and $kp_{q(\ell)}(i)$ may be used in a simple ratio test which limits false matches while maximizing true matches [17]:

Algorithm 3 Lowe Filtering

function `LOWE_FILTERING`($curr$, $prev$)
  for $i = 1...m$ **do**
    if $\frac{d_{H(\ell=1)}(i)}{d_{H(\ell=2)}(i)} < 0.75$ **then**
      $\vec{p}_{np}(i) \leftarrow kp_t(i)$
    else
      discard $kp_t(i)$, $kp_{q(1)}(i)$, and $kp_{q(2)}(i)$
    return $\vec{p}_{np}$

Empirical data suggests applying this simple ratio test and rejecting points with ratios above 0.8 eliminates over 90% of false matches while discarding fewer than 5% of correct matches [17]. The PDF functions demonstrating this behavior are plotted in figure 3·3

Note that newly detected points, $\vec{p}_{np}$, store the keypoints from the *previous* image. This distinction is critical. After point introduction, the `OPTICAL_FLOW` method proceeds to calculate keypoint motion using the LK-optical-flow algorithm. The algorithm uses the previous and current image to determine where the points, $\vec{p}_0$, *from the previous image*, have moved within the current image. So, if points from the current image, $kp_{q(1)}$, were instead stored in $\vec{p}_{np}$, the coordinates would reference incorrect keypoints in the previous frame for all points experiencing motion between frames.

One final note, setup of the feature detector, descriptor and matcher is designed for transparent substitution of algorithms provided by OpenCV. This is useful for benchmarking current algorithms as well as testing detectors, descriptors or matchers developed in coming years.

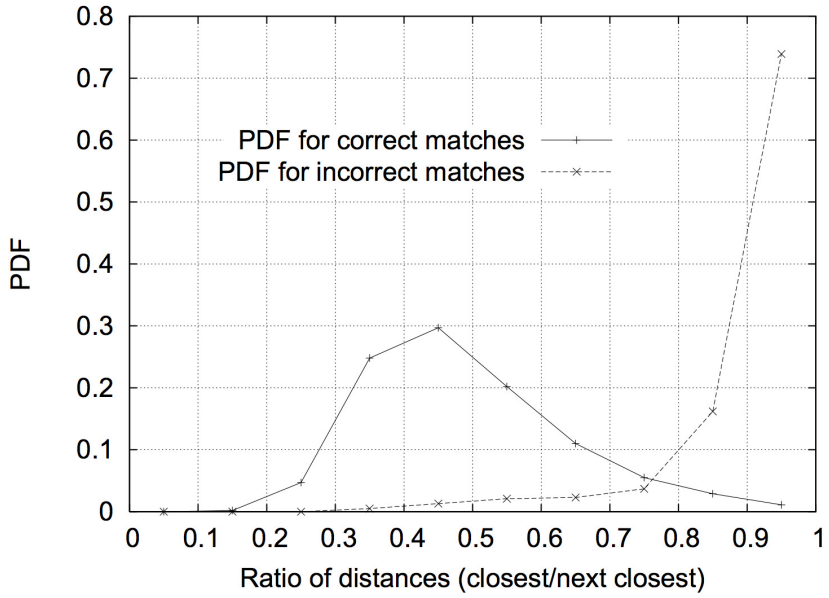


Figure 3·3: Probability that a given match is correct is correlated to the ratio of the nearest to the next nearest keypoint. Data is derived from 40000 keypoints. The solid line represents the PDF for correct matches while the dotted line represents the PDF for false matches. [17]

3.1.2 Point Tracking

Having successfully detected robust keypoints in the image, tracking is performed to ensure persistence of those points while they remain within the frame. The sparse optical flow method suggested by Lucas and Kanade (*section 2.3*) is employed to determine keypoint motion between frames. Using the previous image, *prev*, keypoints detected within *prev*, $\vec{p}_0$, and the current image, *curr*, the method returns the new positions, $\vec{p}_1$ within *curr*. Additionally, a vector of status flags is returned marking points which fell outside of the search window or failed to converge during least squares optimization.

Because LK-optical-flow relies on least squares optimization, forward tracking is somewhat imprecise. The optimization step accounts for minor deviations of the original keypoint area, but it does not resolve accumulated error and, as such, over many calculations, keypoint descriptive regions drift. Take, for example, minor changes

in viewing angle, orientation, or lighting. These changes guarantee differences between keypoint neighborhoods between images. Least squares optimization handles this variation and flexibly tracks points, however, points inevitably adapt, meld and reattach to different objects.

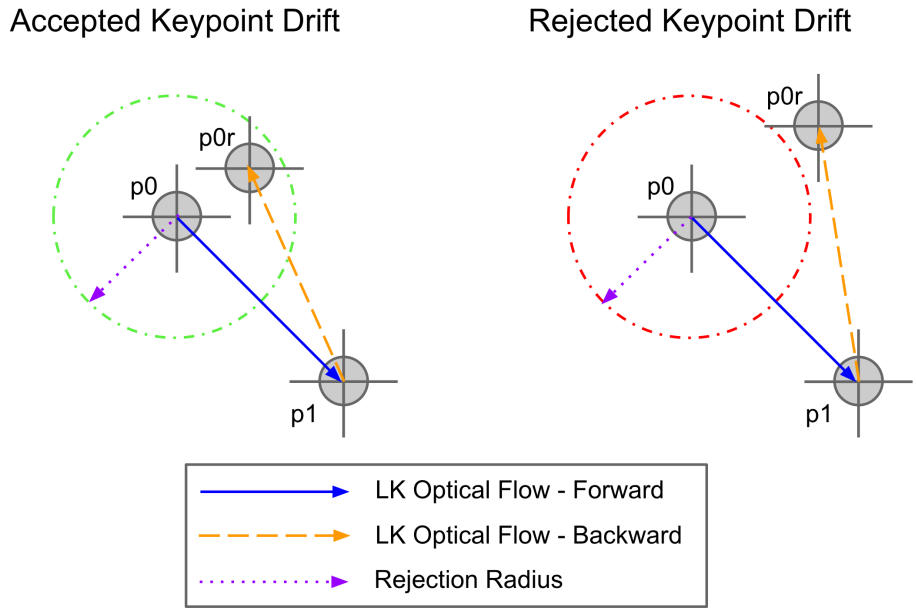


Figure 3·4: Keypoint drift filtering using forward ($p_0 \rightarrow p_1$) and backwards ($p_1 \rightarrow p_{0r}$) Lucas-Kanade optical flow. Rejection radius may be tuned to balance keypoint persistence against keypoint accuracy

As described in the preceding section, the implementation is tuned strongly towards introducing distinct keypoints. Still, when evaluating optical flow, both the keypoint *and* its neighborhood play into the calculation. While the keypoint may be locally unique, it may be part of a larger, densely packed, repetitive pattern, (e.g. a distant chain link fence), texture, (e.g. a stucco wall), or tessellation, (e.g. leaves on a distant bush or tree, or waves in the ocean). Such regions are packed with locally distinct and identifiable features, however the neighborhood surrounding the point is considered to be non-distinct.

Keypoints located in non-distinct neighborhoods are more susceptible to *keypoint drift* than points defined in highly contrasting regions. Still, even highly contrasting

points are affected by keypoint drift. In particular, poor matching performance is rampant in cases where foreground objects obscure background objects. In such instances, points tend to attach to nearby objects and detach from distant ones. This is not entirely undesirable, as looming objects become *weighted* more heavily than distant ones. Still, high levels of drift result in optical flow calculation errors which effect downstream calculations. Keypoint drift must be carefully tuned to minimize induced errors while retaining point adhesion.

One effective solution is performing the LK-optical flow calculation in reverse. Using the current image, *curr*, the newly tracked keypoints, $\vec{p}_1$, and previous image, *prev*, the method re-applies the LK-optical-flow algorithm to determine the back-calculated positions, $\vec{p}_{0r}$, of the recently determined keypoints, $\vec{p}_1$, within *prev*. Points are retained if their back-calculated position lies within a neighborhood around the original points $\vec{p}_0$.

$$\vec{p}_{1f.partial} = \{p_1(i) \mid \quad |\vec{p}_0(i) - \vec{p}_{0r}(i)| < \texttt{MAX\_PXL\_DEV}, \quad i = 1, ..., m\} \tag{3.1}$$

$$\vec{p}_{1f} \subseteq \vec{p}_{1f.partial} \tag{3.2}$$

The general drift reduction process is performed in algorithm 2 lines 11-14 and illustrated in figure 3·4.

Algorithm 4 LK-Drift Reduction and Border Filtering

```
function FILTER_TRIM(p_0, p_0r, p_1)
    b ← [0, ..., 0]
    if |p_0(i) − p_0r(i)| < MAX_PXL_DEV then          ▷ point has low drift
        b(i) ← 1
    if (x_l < p_1(i).x < x_r) & (y_t < p_1(i).y < y_b) then   ▷ point within borders
        b(i) ← 1
    else
        b(i) ← 0
```

Filtering is performed to exclude points with high drift and points exiting the

borders of the image. The `FILTER_TRIM` method is detailed in algorithm 4. It returns a vector mask, $\vec{b}$, with `True` values for points which meet margin and drift requirements and `False` values otherwise. The mask is used to index the vector of raw points, $\vec{p}_1$, and select the final points, $\vec{p}_{1f}$, that are retained from the current frame. x_l and x_r represent the left and right border thresholds respectively (slightly greater than zero and slightly less than image width). y_t and y_b represent the top and bottom border thresholds respectively (slightly greater than zero and slightly less than image height).

Having determined keypoint motions and filtered sub-optimal points, the tracking matrix must be updated to start tracking points which have been introduced, remove records for points which have been filtered, and amend position data for points which have persisted. Per algorithm 2, tracks are created (initialized) in the point introduction section and updated (amended and filtered) in the point tracking section. The tracking matrix, A is composed of elements $p_{(x,y)}(i)_j$ which represent the (x,y) position for the i-th keypoint recorded from the j-th frame (which occurs j frames prior to the current frame).

$$A = \begin{bmatrix} p_{(x,y)}(1)_1 & \dots & p_{(x,y)}(1)_j & \dots & p_{(x,y)}(1)_n \\ \dots & & \dots & & \dots \\ p_{(x,y)}(i)_1 & \dots & p_{(x,y)}(i)_j & \dots & p_{(x,y)}(i)_n \\ \dots & & \dots & & \dots \\ p_{(x,y)}(m)_1 & \dots & p_{(x,y)}(m)_j & \dots & p_{(x,y)}(m)_n \end{bmatrix} \tag{3.3}$$

The `CREATE_TRACKS` process appends rows to the tracking matrix for each newly discovered point. The process is performed during point introduction to ensure equal cardinality between the rows of the tracking matrix and the reconstructed (after point introduction) keypoint vector, $\vec{p}_0$. This guarantees that the i-th element of $\vec{p}_0$ corresponds to the i-th row of the A matrix.

The `UPDATE_TRACKS` process removes track histories for points which have been

filtered out, and updates position data for all remaining points. The A matrix is pruned of the filtered keypoints as follows

$$A_{+1} = BA \quad B = diag(b) \quad \text{s.t.} \quad B \ \text{ is } \ m \times m \tag{3.4}$$

The submatrix, A_{+1}, corresponds to tracking information for the set of points p_{1f} that were not filtered out. B is the diagonal matrix created with the vector mask $\vec{b}$ whose elements are 0's for filtered points and 1's for retained points. The cardinality of the rows of A_{+1} equals the length of p_{1f} or the sum($\vec{b}$), and again the i-th element of $\vec{p}_{1f}$ corresponds to the i-th row of A_{+1}. Next, individual tracks are updated to include recent keypoint motions. This is done through column insertion. Namely, the vector p_{1f} is inserted into the first column of A_{+1} and the n-th column is removed. Newly introduced tracks, however, have fewer historic coordinates, and do not have points to fill the entire row. This could be solved by initializing the row with default *filler* values which are pushed out as coordinate history becomes available. The default value could be any point outside the image window size, e.g. $(10w, 10h)$. This way the default points may be recognized and are very noticeably outside of the image window.

A clever trick simplifies the implementation of the tracking matrix in code dramatically. The heart of the structure is a standard FIFO styled array called a `deque`, found in the `collections` library. `Deque`'s are versatile buffers with selectable length that automatically shift and pop elements as necessary. A `deque` is created for each newly introduced keypoint and added to a list (python version of an array). After the point mask, $\vec{b}$ is determined, the list of `deque` objects is pruned of non-persisting points. Finally, the structure is updated with tracked points by *pushing* values onto the associated `deque`. As the structure is composed of standard storage types, it may be treated as a nested list of lists, making it highly versatile. For instance, plotting

each track individually is achieved by passing the entire list structure directly into OpenCV's `polylines` function.

The `image` library offers rapid and robust processing of optical flow. A number of visualization functions are available but are not detailed for the sake of brevity. With robust optical flow available, processing may begin in order to guide and control vehicles using video streams.

3.2 Control

Feedback control using robust image features can be simple and intuitive. The image plane derives from sensors fundamental to human exteroception, and for most, is innate to our very being. The image plane deviates from our own perception in two primary ways: a dimensional reduction from 3D object fields to 2D planar images, and our experience based, association library which quickly segments the object field into recognizable or definable forms.

The mathematical reduction from spatial fields to planes can, in many cases, be circumvented, and acclimation to the alternate perception occurs rapidly. Any person who has watched a television has experienced this acclimation. Still, the spatial reduction results in information loss, transforming certain unique 3D perceptions into 2D multiplicities. One example is distinguishing between the 2D optical fields caused by pure lateral translation and pure yaw.

The absence of (or attempt to decouple our) historical object associations is more challenging. Remaining objective when inspecting optical fields plotted on an image is difficult. Equivalently, observing an optical vector field without its corresponding image, inferring the scene, and resolving the appropriate control is practically impossible and, at best, confounding. As such, balancing quantitative analysis and qualitative observations is critical when designing and implementing image feedback

control laws.

While descendant from sound theory, the flow balancing strategies implemented by this paper are largely empirical; still, their practical viability is clearly evident. In short, the transition from practicum to practiced—handling massive and noisy data sets, real world constants and delays, discretized versus continuous data—is rife with experimentation and intuition.

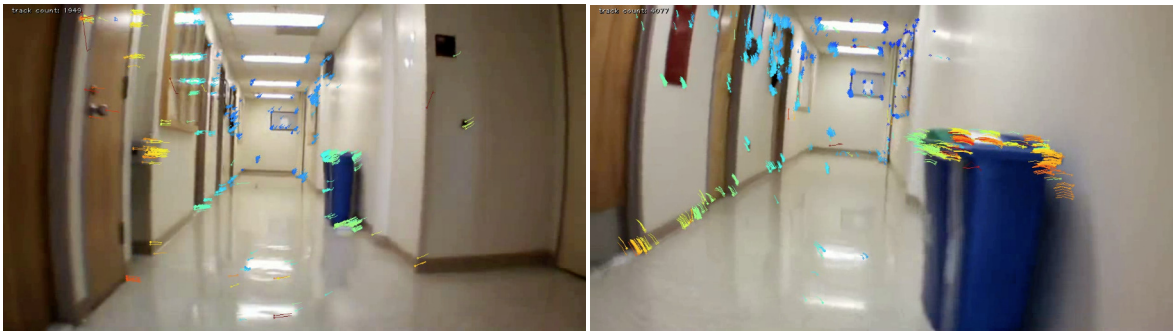
3.2.1 Signal Inputs

The control system was implemented on an aerial vehicle, specifically a quadcopter. Quadcopter dynamics, internal control loops and sensors are very widely studied and therefore left as an exercise for the reader. Two concepts are worth restating for the reader. Firstly, a quadcopter can move in six degrees of freedom and specifically, has three linear motion axes and three rotational axes. Using common conventions, the vehicle can move rotationally about pitch, roll or yaw axes or move translationally, along lateral, longitudinal or vertical axes. Secondly, a quadcopter is an underactuated vehicle, meaning, while it may travel in six degrees of freedom, it must do so using only four inputs: thrust, pitch, roll and yaw. The critical point is that for quadcopter to move longitudinally or laterally, it does so by applying pitch or roll. Less intuitively, quadcopter yaw is inherently linked to thrust and as such, applying thrust induces yaw and vice versa. Despite this underactuation, when considering stabilized, steady-state motions (having completing the transient acceleration to tilt the vehicle) each motion primitive is associated with a distinct optical flow field. Like most engineering systems, compound system inputs result in compound system outputs which add through superposition. Similarly, when observing system output, compound motions may be broken into component-wise motion primitives

Figures 3·5, 3·6, and 3·7 demonstrate optical flow fields for the three primary linear motions in the longitudinal, lateral and vertical directions. Given a vehicle

(a) Forward moving (b) Backward moving

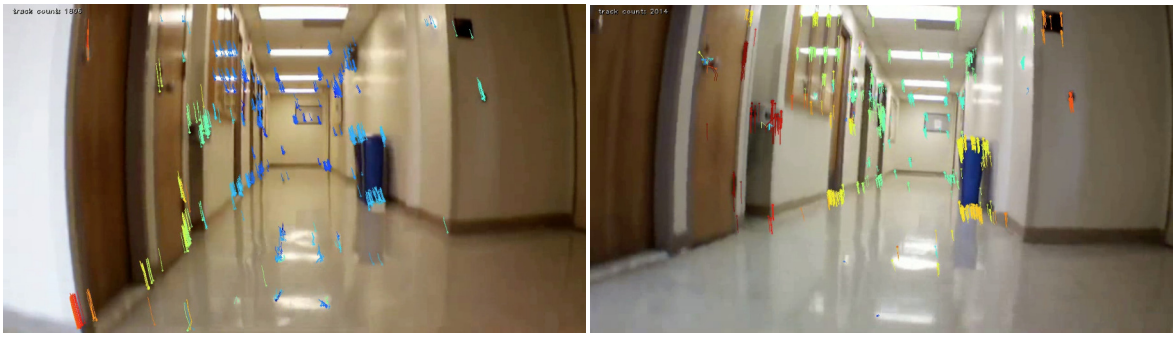
Figure 3·5: Optical flow field for (a) advancing and (b) retreating quadcopter. Note how objects in the foreground are associated with larger flows than those in the background. Forward and backward motions are inherently linked to pitch commands for quadcopters, however at steady state, the primary result is a camera pointed downward (advancing) or upward (retreating).



(a) Leftward moving (b) Rightward moving

Figure 3·6: Optical flow field for (a) leftward and (b) rightward translating quadcopter. Note how objects in the foreground are associated with larger flows than those in the background. Translation is linked to roll commands for quadcopters and hence, influence the flow field during transient motions. A minor rotational effect is evident and is superimposed onto the image plane during the maneuver.

with a forward facing camera, divergent or convergent optical flow fields centered about the focus of expansion, indicate the vehicle is traveling forwards or backwards respectively. On the other hand, a near uniform flow field where flow moves across the image plane to the left or to the right indicates that the vehicle is translating to



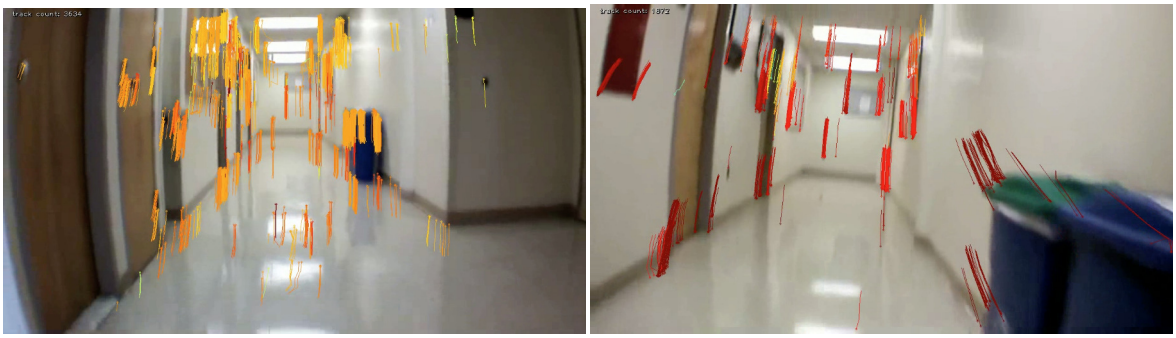
(a) Ascending (b) Descending

Figure 3·7: Optical flow field for (a) ascending and (b) descending quadcopter. Note how objects in the foreground are associated with larger flows than those in the background. Ascent and descent inherently induce minor yaw rotations for quadcopters, however this particular vehicle is quite stable and the transient effect is not noticeable.

the right or to the left respectively. Equivalently, a near uniform flow field where flow moves upwards or downwards indicates descent and ascent respectively.

One critical defining feature of translationally induced flow fields is that object depth is directly proportional to optical flow. Observing figures 3·5, 3·6, and 3·7 a second time, it is clear that foreground objects are associated with faster moving keypoints (indicated by reds and oranges), while background objects are associated with slower moving keypoints (blues and greens). This distinction is critical when comparing translational motions against yaw and pitch rotations.

Figures 3·8, 3·9, and 3·10 demonstrate optical flow fields for the three primary rotational motions: pitch, roll, and yaw. Given a vehicle with a forward facing camera, a uniform flow field where flow moves across the image plane to the left or to the right indicates that the vehicle is yawing to the right or to the left respectively. Equivalently, a uniform flow field where flow moves upwards or downwards indicates downward and upward pitch respectively. Lastly, a vehicle applying a clockwise or counterclockwise roll results in a a flow field whose curl is counterclockwise or clockwise respectively. In all cases, the magnitude of the vector field (or curl) is directly proportional to the



(a) Forward tilting (b) Backward tilting

Figure 3·8: Optical flow field for (a) forward and (b) backward pitching quadcopter at the begining of a longitudinal motion. Flow field is nearly uniform.

(a) Right leaning (b) Left leaning

Figure 3·9: Optical flow field for (a) rightward and (b) leftward rolling quadcopter at the begining of a lateral motion. Flow field has a high level of well defined curl.

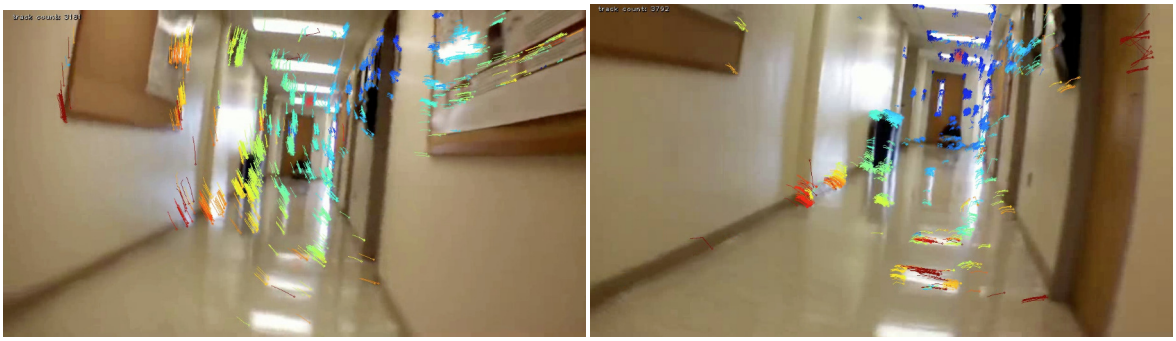
rate of rotation.

As previously mentioned, the yaw and pitch induced optical fields are almost perfectly uniform. Optical flow is therefore independent of object distance, $x(t)$. Furthermore, because of the high leverage that the vehicle has over the camera, optical fields generated by rotation have higher signal gain than translational motions. Again, this attribute cements the importance of designing controllers or signals which are able to compensate, filter, or ignore rotationally induced fields.

Applying any of the six motion primitives in conjunction results in a knitting of

(a) Left turning (b) Right turning

Figure 3·10: Optical flow field for (a) leftward and (b) rightward yawing quadcopter. Flow field is nearly uniform.



(a) Leftward translation following CCW roll (b) Leftward motion while moving forward

Figure 3·11: Optical flow fields for two compound motions. Deciphering the motion from the optical flow field can be challenging without additional sensory input.

their induces optical flow fields. The fields add through superposition, and as mentioned above, decomposing compound motions in their component motion primitives is a significant challenge—especially when distinguishing between yaw and lateral motions and between pitch and vertical translations. In general, decoupling translation and rotation induced flows is a well-known problem in computer vision ([29]) that is beyond the scope of the present work.Having defined system inputs, outputs and observer measurements, the focus may switch to the topic of control.

3.2.2 Control Overview

The system feedback control law applies a bisection strategy to balance and guide the vehicle evenly between nearby and looming objects. Time-to-transit is computed for each keypoint located within the frame. The image plane is divided into three equally sized vertical panels. Time-to-transit is averaged over each of the outer panels using only points found within that panel. The control signal is created using the average τ value determined within each panel by applying the method described in algorithm 5.

Algorithm 5 Time-to-transit Feedback Control

if $\text{sgn}(\tau_L) \neq \text{sgn}(\tau_R)$ **then** ▷ uniform flow direction
  $motion \leftarrow \text{sgn}(\tau_L)$ ▷ $\tau_L < 0$ leftward or $\tau_L > 0$ rightward
  $signal \leftarrow motion \times (\tau_L^{-1} + \tau_R^{-1})$
else ▷ divergent flow
  $motion \leftarrow \text{sgn}(\tau_L)$ ▷ $\tau_L < 0$ backward or $\tau_L > 0$ forward
  $signal \leftarrow motion \times (\tau_L^{-1} - \tau_R^{-1})$

Here, τ_L and τ_R represent the averaged time to transit values for all keypoints in the left and right panel respectively. The variable, *motion*, indicates the direction of motion determined using tau_L although it could equivalently be determined from the negative of tau_R. The sign of left panel and right panel time-to-transit indicates flow field type—divergent or uniform—as well as the vehicle motion direction.

The critical quality discussed by *section 3.2.1 Signal Inputs* was that because rotational operations induce uniform, high gain signals, those signal components must be compensated, filtered, or ignored. By careful tracking of *tau* sign and vehicle motion direction, the differencing strategy inherently compensates for high gain coming from uniform flow fields. While imperfect—rotational flow fields skew slightly with motion—the differencing strategy and the resulting signal have proved viable in practice. Other rotational considerations are discussed later in the section.

The signal from algorithm 5 is used by the vehicle motion controller to apply evasive or corrective motions. This strategy is summarized in algorithm 6.

Algorithm 6 Vehicle Signal Application

```
if signal < 0 then
    MOVE_LEFT
else
    MOVE_RIGHT
```

A visual strategy was developed to demonstrate how algorithms 5 and 6 apply controls using the time-to-transit signal and more specifically, its inverse, loom. The summary of possible system inputs and outputs are displayed in figure 3·13 and figure 3·14 for divergent and uniform flow fields respectively. This strategy successfully guides a vehicle along a corridor, avoiding walls, or other similar obstacles that become *significant* within the visual field.

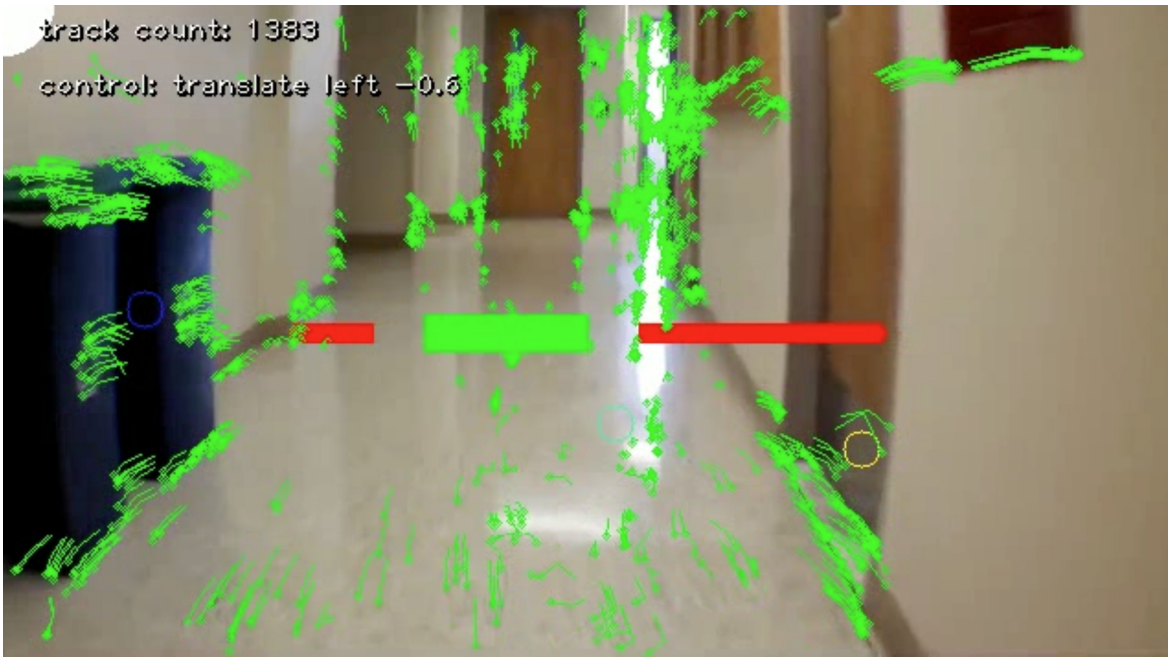


Figure 3·12: Control system visualization demonstrating signals applied by vehicle transiting a hallway

One fundamental assumption the control strategy requires is that the focus of

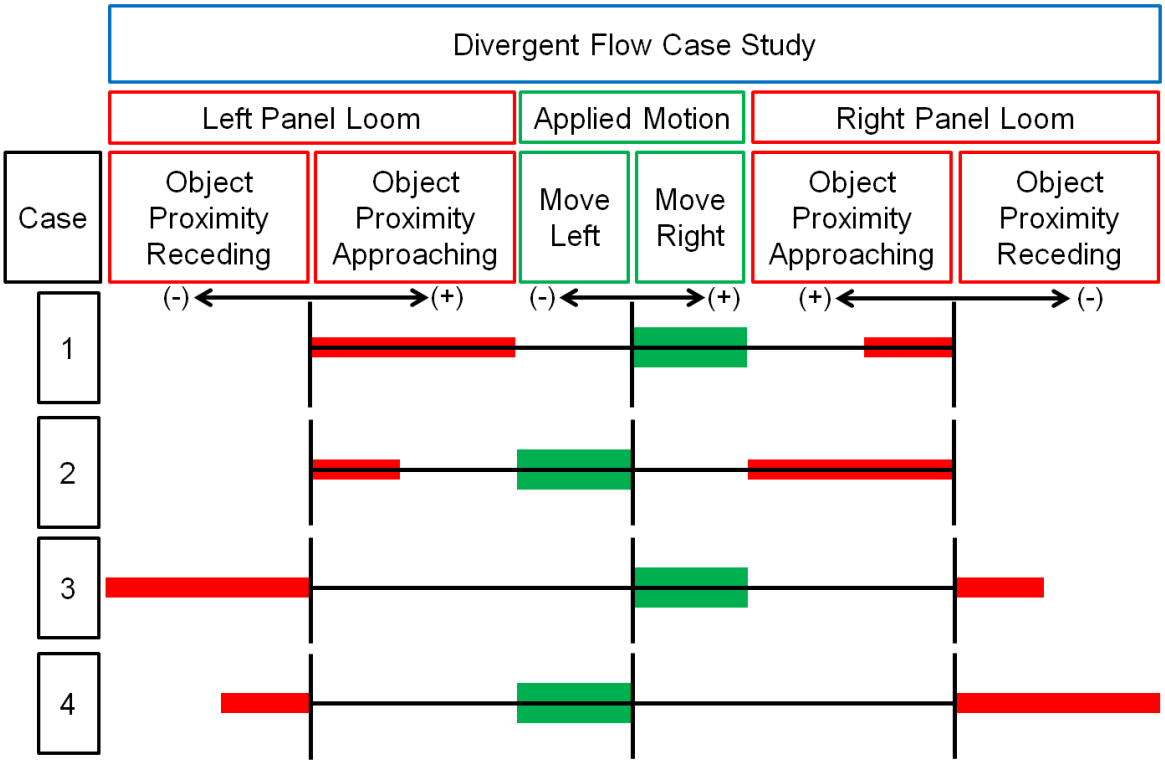


Figure 3·13: Controller response for divergent flow TTT values. The evasive actions performed for each of the cases depicted above are as follows:

1. Forward moving vehicle, object looming on left. Control applies movement right.
2. Forward moving vehicle, object looming on right. Control applies movement left.
3. Backward moving vehicle, object receding rapidly on left. Control applies movement right.
4. Backward moving vehicle, object receding rapidly on right. Control applies movement left.

expansion lies at the center of the frame. Time-to-transit values are calculated by first centering the coordinate system at the center of the image and determining distance d_i from the origin. The assumption remains valid during forward flight, when forward velocity is greater than lateral, vertical or rotational components. Compound motions

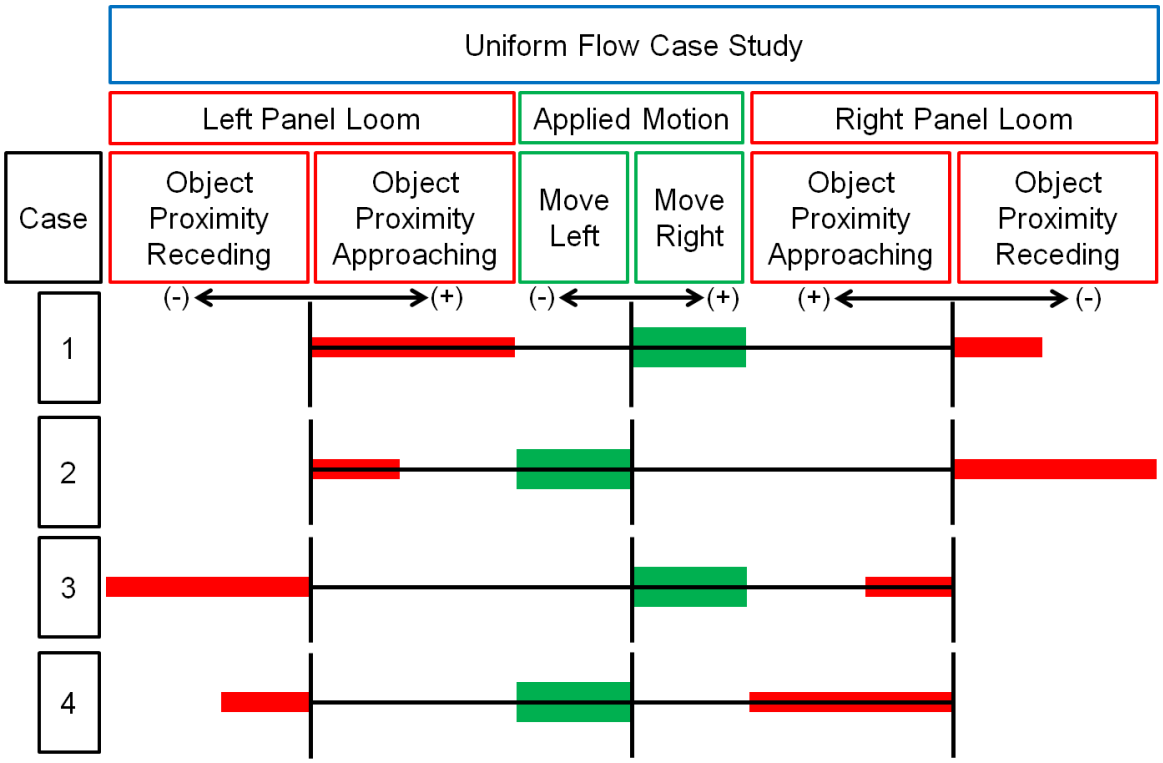


Figure 3·14: Controller response for uniform flow direction TTT values. The evasive actions performed for each of the cases depicted above are as follows:

1. Leftward moving vehicle, object looming on left. Control applies movement right.
2. Leftward moving vehicle, object receding on right. Control applies movement left.
3. Rightward moving vehicle, object receding on left. Control applies movement right.
4. Rightward moving vehicle, object looming on right. Control applies movement left.

cause the camera axis to skew from the vehicle direction of motion, which results in a shift of the focus of expansion. One calculation which could mitigate skew induced errors would be to determine the location of the focus of expansion and calculate distance values, d_i, relative to the FOE. An attempt was made to implement this

feature, however a robust calculation to locate the FOE proved quite challenging.

As mentioned in *section 2.2 Time to Transit*, points within ϵ pixels of the image center often result in near-zero time-to-transit values suggesting those points will soon cross the image plane. The center point of the image is in fact a singularity for the control system because d_i is always near zero. In actuality, these points are almost always false indications. To prevent introducing error into the control strategy, points located in the center of the image (and for simplicity, the center panel) are not used. Future design of image panel partitions could improve overall image utility, which is discussed briefly in *section 5.2 Future Work*.

The singularity near the center of the image plane may be avoided by rotating the camera axis ninety degrees from the axis of forward motion. One example is demonstrated in the work performed by Kong et al [1]. The side-viewing configuration shifts attention from object looming (collision considerations) to object transit (vehicle guidance). It also offers powerful and implicit geometric prioritization and compartmentalization of keypoints within the image. Stated explicitly, for side-facing configurations on forward—or even slightly off axis—moving vehicles, keypoints near the center of the frame are *always* attached to objects being transited by the vehicle, while points near the edges of the frame are *always* points looming towards or away from the vehicle. This eliminates many of the uncertainties associated with 2D multiplicities characteristic of a forward facing camera.

3.2.3 Control Implementation

To maintain near real time performance, all control calculations are performed using python's NumPy library. Further building on *section 3.1 Image Processing*, the output vectors, $\vec{p}_{1f}, \vec{p}_{0f}$, from algorithm 2, which each contain m keypoint (x_i, y_i) coordinate pairs are leveraged to perform every control calculation. Firstly, consider

the form of $\vec{p}_{1f}$ and $\vec{p}_{0f}$ as follows:

$$\vec{p}_{0f} = \begin{bmatrix} x_{0f}(1) & y_{0f}(1) \\ ... & ... \\ x_{0f}(i) & y_{0f}(i) \\ ... & ... \\ x_{0f}(m) & y_{0f}(m) \end{bmatrix} \quad \vec{p}_{1f} = \begin{bmatrix} x_{1f}(1) & y_{1f}(1) \\ ... & ... \\ x_{1f}(i) & y_{1f}(i) \\ ... & ... \\ x_{1f}(m) & y_{1f}(m) \end{bmatrix} \tag{3.5}$$

Keypoint coordinate pairs, $(x_{jf}(i), y_{jf}(i))$, are given in the pixel coordinate system depicted in figure 3·2. Images sizes as implemented in software, are defined by a width and height in pixels. Time-to-transit calculations require distance from the FOE, which, under the assumption made by the control law, is approximated at the center of the image. For an image of width, w pixels and height, h pixels, the image center is located at:

$$(w_c, h_c) = \left(\left\lceil \frac{w}{2} \right\rceil, \left\lceil \frac{h}{2} \right\rceil \right) \tag{3.6}$$

The coordinate system is shifted to the image center by performing:

$$\begin{aligned} x_c &= x_{1f} - w_c \\ y_c &= y_{1f} - h_c \end{aligned} \tag{3.7}$$

Optical flow vector x and y components between adjacent frames are calculated in Cartesian coordinates as

$$\begin{bmatrix} \frac{\partial x_c}{\partial t} & \frac{\partial y_c}{\partial t} \end{bmatrix} = \frac{\vec{p}_{1f} - \vec{p}_{0f}}{\partial t} \tag{3.8}$$

Using a change of variables, optical flow may also be calculated in polar coordinates using the following equations:

$$\begin{aligned} d &= \left\{ \sqrt{x_c(i)^2 + y_c(i)^2} \quad s.t. \quad i = 1...m \right\}, \\ \frac{\partial d}{\partial t} &= \left\{ \frac{x_c(i)\frac{\partial x(i)}{\partial t} + y_c(i)\frac{\partial y(i)}{\partial t}}{d(i)} \quad s.t. \quad i = 1...m \right\}. \end{aligned} \tag{3.9}$$

Finally, time-to-transit may be calculated in two different forms; Cartesian and polar.

$$\tau_x = \left\{ \frac{x_c(i)}{\frac{\partial x_c(i)}{\partial t}} \quad s.t. \quad i = 1...m \right\} \qquad x - \text{component of T.T.T,} \tag{3.10}$$

$$\tau_y = \left\{ \frac{y_c(i)}{\frac{\partial y_c(i)}{\partial t}} \quad s.t. \quad i = 1...m \right\} \qquad y - \text{component of T.T.T,} \tag{3.11}$$

$$\tau_d = \left\{ \frac{d(i)}{\frac{\partial d(i)}{\partial t}} \quad s.t. \quad i = 1...m \right\} \qquad \text{radial form of T.T.T.} \tag{3.12}$$

The purpose of calculating time-to-transit into both Cartesian and polar coordinate systems, is that each result may be used for different forms of control.

The component breakdown of time-to-transit into the Cartesian system, allows one to apply the simplifying assumption that the vehicle moves within a plane and is not concerned with objects above or below the vehicle. In such an instance, τ_x, indicative of lateral flow, is the only signal of concern for flow balancing. Maintaining balance per section *3.2.2 Control Overview* is simplified because pitching rotations (see figure 3·8) have little to no impact on τ_x. Yaw has a much larger effect on τ_x signals, however, yaw is not critical when flying down a corridor and can generally be assumed near zero.

Pitch rotations are incidental when performing forward motions and forward motions are integral to the controller; τ_y is strongly impacted by pitching motions and hence the signal may be strongly effected by error input. By ignoring the τ_y signal, a significant noise contributing component impacting system signal-to-noise is eliminated.

Still, one could attempt to incorporate τ_y into the control scheme if they first considered how to handle pitching motions. Separate feedback controls could apply ascent and descent commands to avoid obstacles within a second dimension. This

would offer alternative routes for object avoidance. τ_y control might also be powerful when balancing a vehicle vertically within a corridor or structure. Regardless, before implementing such a strategy, further research and development is necessary to identify, partition and manage rotationally induced optical flow vector.

The radial calculation of time-to-transit, τ_d, determines the rate at which a keypoint travels towards or away from the image plane while ignoring angular motions. This strategy augments control feedback particularly well for a forward moving quadcopter that applies lateral motions to balance flow. A forward moving vehicle is associated with divergent flow fields (see figure 3·5) which may be reliably balanced to avoid looming objects. Additionally, as demonstrated in figure 3·9, rolling motions—intrinsic, transiently, to lateral motion for a quadcopter—induce rotational flow fields. By dropping the angular component of τ, the control law ignores confusing transient effects induced directly by the output of the control. This is important for maintaining observer validity during flights. τ_d and τ_x controls were both implemented and tested successfully on board the vehicle. The test results are discussed in further detail in *Chapter 4 Experimental Results*

In order to average time-to-transit for prescribed grid geometries, a system was developed to locate each keypoint within an encompassing grid on top of the image. While the current implementation segments the image into a tryptic (one row, three columns), a generic two-dimensional histogram function was created to determine grid space number, $i \times n + j$, for an arbitrary point located in the i-th row and j-th column of a grid divided image plane composed of m-rows and n-columns. Indices i and j are zero-based and range to $m - 1$ and $n - 1$ respectively. The grid space numbers may be quickly checked and converted into a boolean index to select all points located in a single cell on the grid. The boolean index may also be used to quickly select and average τ values for each individual cell. Inverting the τ average

generates the value for loom which is used directly by the control system. For further details, this function, `bin_indexing2d`, may be examined in the code section of the Appendix.

The control implementation includes one final assumption, namely that τ values above a given threshold may be ignored in order to eliminate background noise. After performing 2D histogram binning, a threshold is applied to τ values which filters time-to-transit values higher than a fixed threshold. The selected threshold, ten seconds, was determined empirically using observations from testing. By ignoring points with τ greater than ten seconds, the cell averages avoid false weighting of distant points whose τ values may approach infinity. The resulting average is a closer representation of points the vehicle must attempt to avoid. A similar issue exists on the opposite spectrum for points which transit the image plane (often introduced through mismatching), although an argument exists for the case described by figure 2·4 for moving observers and moving objects. These points may transit the image plane rapidly enough that the vehicle should not and could not attempt to respond. Recent work by Professor John Baillieul regarding 'Geometric Feature Saliency in Optical Flow', indicates that a biological precedent exists to ignore features moving beyond a certain rate in the image plane. This feature was implemented in code, however, because of the asymptotic form of time-to-transit—ranging from 0 to infinity—averaged values are less strongly influenced by points near zero than those with higher values. Finding an empirical cut off was difficult and the feature was temporarily turned off.

The final stage before implementation onto the quadcopter was developing a vehicle interface able to receive and process vehicle data and apply commands based on the control law. The AR.Drone platform proved the perfect testing ground on which to prove the control laws presented.

3.3 Platform

3.3.1 AR.Drone2

The Parrot AR.Drone 2 is a commercially available, robust, and pre-built quadcopter system. It is constructed with a variety of onboard sensors optimal for testing the control architecture and image processing system. Sensors include a 6-DOF IMU, downward facing ultrasonic range finders, and forward pointing and downward pointing cameras. While data was only explicitly taken from the forward pointing camera, the inner control loop of the AR.Drone2 uses all of its sensors to maximize positional precision during operation and avoid unwanted drift.

Figure 3·15: Parrot AR.Drone2

The vehicle can be controlled using one of many commercially available applications developed for standard tablets and smart phones. Parrot also offers a software development kit (SDK) to its developers, to promote third party enhancements. This system is unfortunately not well tuned towards integrating control systems, inclined instead towards user interface development and user control. To overcome this, ROS

(Robot operating system) created a platform which uses the Parrot SDK and simplifies implementing system level control. Unfortunately, the coupled operating system is difficult to set up, clunky, and ultimately an impediment to itself and new users.

Python SDKs were similarly scarce, the only available system was an open source project developed under the MIT License by Bastian Venthur for the AR.Drone 1 [30]. The open source code is pithy and simply performs all primitive communication procedures with the drone, primarily socket communications to enable the vehicle, control its motions, and receive and decode sensor and video data packets.

The python SDK was selected for its simplicity and readability. A top level system was developed to simplify vehicle control and create a modular interface to *plug in* and test generic controller implementations created for the project. The system enables rapid switching between feedback control and user control. A computer keyboard acts as the user interface to the vehicle. A number of hot-keys were generated to quickly reset, record images, save video and display information to the user. Further details explaining the top level controller are written into the code comments. Code for the UI and integrated controller are presented in the software appendix.

3.3.2 Thread Handling

To achieve the best possible performance by the software system, multi-threading was applied for all IO (input-output) bound processes. Unlike other languages, python threading does not perform true parallel processing. Python threads are controlled by a global interpreter lock (GIL) which must be possessed by the *thread* under current execution. The result is a linearly run system which has the capability to wait when data is not available to process. This does not improve program performance when executing processor bound, computationally intense routines, however, it is incredibly useful when waiting for data from ports or sockets (AR.Drone2 data and control sockets), saving files (all video saving operations), displaying items to the UI

(transferring images to the GPU and plotting them on the screen), or incorporating delays to allow system commands to propagate through the vehicle. While waiting for threads to complete IO bound processes, the system performs all necessary numerical activities including calculations for keypoint introduction and tracking, optical flow, time-to-transit, histogram binning and control application. The numerical calculations complete before the next image is available for processing. All thread communication occurs using locks, events or queue objects; shared resources are all held briefly and handled carefully to avoid race conditions. True parallel processing is possible in python and may be performed using the multi-processing library. The library was avoided in anticipation of deployment onto single processor systems prevalent on most robotic vehicles.

Chapter 4

Experimental Results

4.1 Software Performance

A number of user controlled flights were executed to gather flight video data for testing. The videos were fed into the image processing software to determine system performance. Algorithm sensitivity was determined against the primary variable: the number of tracked features. The computer used for software testing is a 2013, 15" Macbook Pro with an Intel i7 Processor at 2 GHz, 256 KB of processor L2 cache (per core), 6 MB of L3 cache and 8GB of ram. All image processing, control and plotting was performed on the CPU on a single core. Flight systems and communications for the AR.Drone2 were controlled on a separate core. The results of the image processing and control scheme are summarized by figure 4·1.

Each code section listed in the legend is for the explicit action indicated by the name. This implies all data or actions required to perform a specified task have previously been executed i.e. *matching* corresponds to the time required for matching keypoints assuming detection and description have already happened and the results are available for use.

A number of code sections barely register time delays for the system; the computation time also scales rather slowly against the quantity of tracked points. In 4·1, code sections with processing times less than 0.5 ms are listed in the order of execution: Lowe-filtering (filtering newly matched keypoints based on their distance ratio), point insertion (adding new keypoints to the tracking matrix), track-updates (prun-

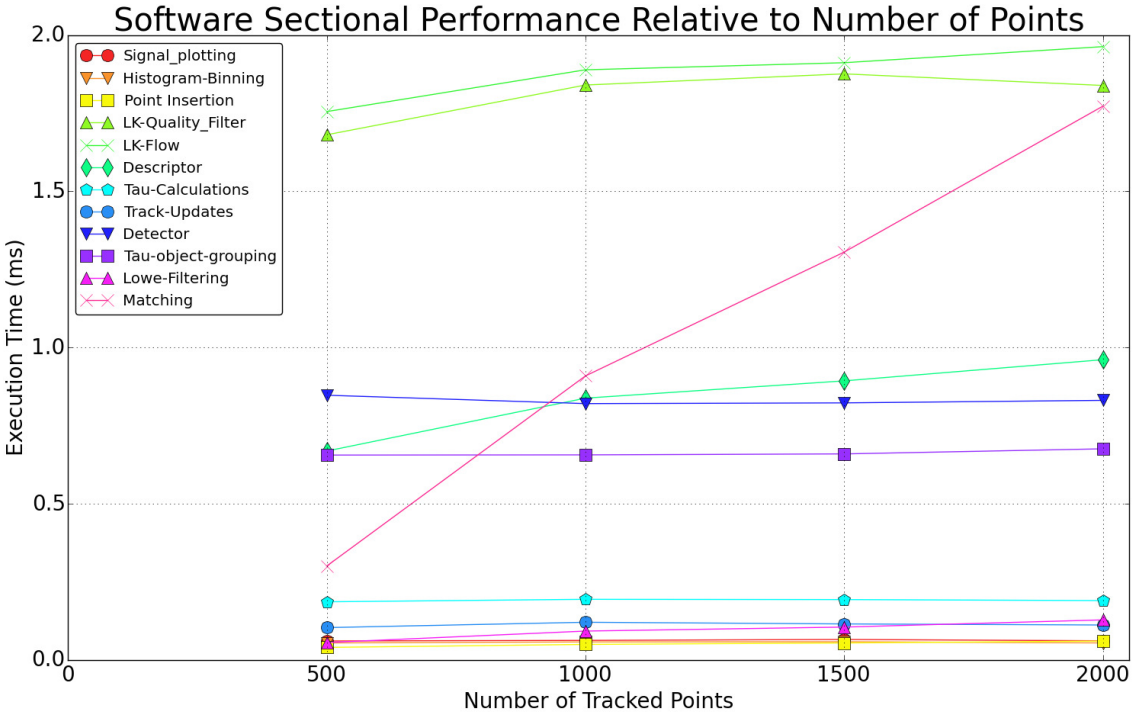


Figure 4·1: Algorithm time sensitivity to number of tracked keypoints.

ing the tracking matrix of filtered points), histogram-binning (locating and indexing points within the prescribed grid, a tryptic in this case), tau-calculations (calculating time-to-transit), and signal_plotting (plotting the control signal ONLY).

Two sections have execution times which are nearly constant with number of tracked points. These sections fall in the 0.5-1.0 ms range and include: detection (FAST detector should be nearly constant based on the size of the image), and tau-object-grouping (a simple object tracking strategy discussed in further detail in section *5.2 Future Work*. The primary time-dependence stems from the prescribed image grid, a tryptic in this case).

Code sections with execution times greater than 1 ms account for the majority of algorithm execution time. In order from least to most time consumed: descriptor (Computing the FReaK descriptor), matching (performing brute-force matching; each keypoint is tested against every other, a handshaking problem that scales quadrat-

Keypoints (#)	500	1000	1500	2000
Total Rate (Hz)	156	133	124	116

Table 4.1: Image processing and control system total algorithm processing rates

ically with number of points), LK-Quality_Filter (Lucas-Kanade optical flow in reverse), and LK-Flow (Lucas-Kanade optical flow in the forward direction). The worst scaling challenge comes from the brute-force matching algorithm. Some of this is mitigated by selectively introducing points from FAST into regions requiring keypoints. Still, even this strategy does not guarantee robust points and many may be filtered during matching for minimal performance gains.

Overall, the image feedback control calculation rates were very promising. All processing occurred well above the vehicle frame rate of 30 Hz. Table 4.1 summarizes the software rates for the control system, including image processing through control signal output.

In flight testing, plotting was the most time consuming task, but only when plotting full keypoint tracks with color adaptations to indicate optical flow rate or time-to-transit. While real-time plotting was very useful during testing, it required significant time and reduced the algorithmic rate to 30 Hz, the same rate at which the vehicle transmitted frames. Simpler plotting strategies can work around this minor hurdle.

4.2 Vehicle Performance

Early in development, it was determined that the python AR.Drone2 implementation had significant latency when sending frames to the controlling computer. Testing indicates the latency stems from decoding the h.264 stream output by the drone using the industry standard tool FFmpeg. A number of attempts were made to reduce the latency including decoding programs written in C, C-based implementations of the

AR.Drone2 code (changing platforms) and python speed up options (iron python, cython, psycho). Despite this, a 0.5 second latency remained throughout testing. Latency was much improved for the AR.Drone2 source SDK provided by Parrot, at 0.2 seconds. Unfortunately, aligning this SDK with the python image processing was both challenging and time consuming causing a required shift to maintain primary project objectives. To mitigate the large latency, the AR.Drone2 was flown at slowed speeds to give the control ample time to react to vehicle motion.

4.3 Flights

The software architecture and control law were tested on board the Parrot AR.Drone2. Flight tests included a variety of locations of two basic compositions: indoor flights through a hallway, and outdoor flights along a road. Some test flights (outdoors) included the extra margin of safety provided by attachment to a tether. At the completion of work, the vehicle would reliably traverse the length of the hallway or road while simultaneously avoiding looming objects keeping the path centered. A number of successful test flights are illustrated in the figures below.

Image size from the AR.Drone2 was not varied and all testing took place for a frame size of $(640, 480)$. Point tracking and point introduction worked flawlessly. The system reliably captured important image features, tracked those features through time, dropped keypoints as they exited the frame or accumulated error and finally introduced new robust points quickly as needed. The system was tested successfully using a range of tracked keypoints from 500-2000. Ultimately, 1000 keypoints were selected as the optimum balance between maintaining enough points to define feature poor regions while preventing over definition of feature rich areas. Additionally, the reduction of matching computation time was considered worthwhile.

The vehicle performed as desired in indoor and outdoor flights. One observable

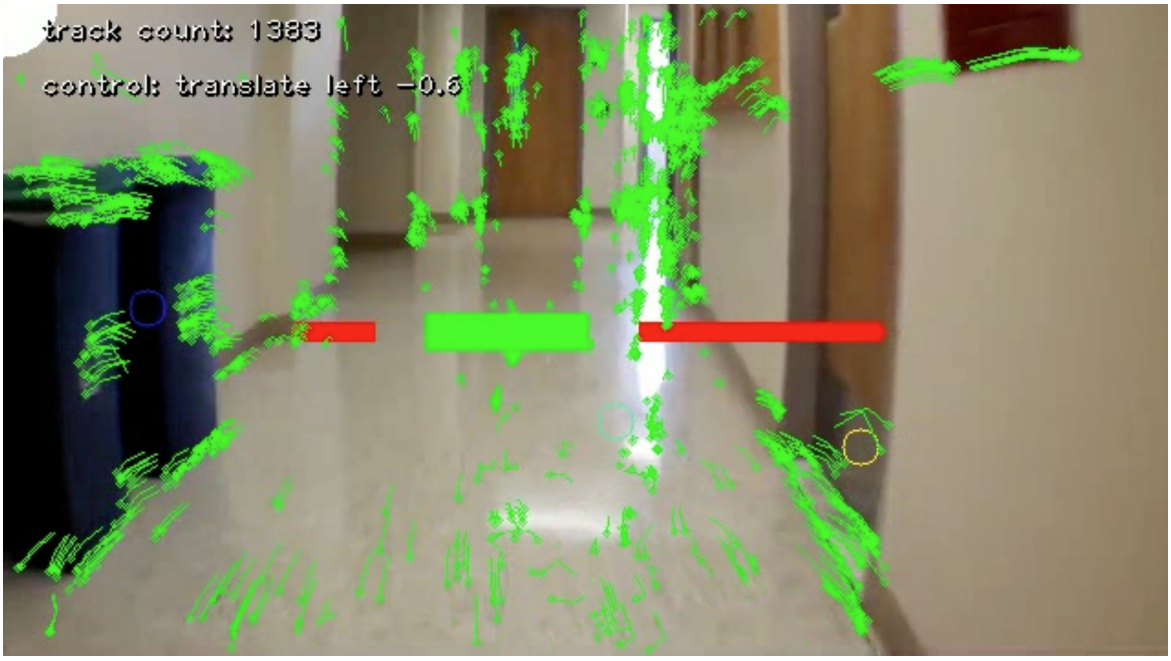


Figure 4·2: Successful indoor flight navigating along a hallway.

Figure 4·3: Successful outdoor flight navigating along a roadway.

difference between the two environments was the behavior of the vehicle when traversing a narrow path as opposed to a wide open one. Flights along a narrow path were

characterized by a centering behavior between the two walls or looming objects. Alternatively, flights along a wide path resulted in a zig zagging behavior, where the vehicle shifted away from looming objects until approaching the opposing barrier.

Figure 4·4: Succesful navigation of hallway with high initial skew from motion direction.

Successful testing was also performed with high skew angles between the camera axis and the direction of motion. Before flight initiation, the vehicle was yawed to point between the axis of the hallway and the left-side or right-side barrier. The vehicle compensated by performing higher percentages of lateral motions away from the barrier, resulting in a canted, diagonal flight along the hallway. These results are promising for a number of reasons. Firstly, the high skew suggests the control algorithms' resilience to initial conditions. Secondly, the canted flight was a successful demonstration and implies the great potential for flight configurations and control as suggested by [1]. Lastly, the successful high skew testing indicates errors induced by off angle flight, may be less severe than originally anticipated.

Overall, flight testing was remarkably successful and the results show great promise

for the software platform, test vehicle, and the basic control laws which were implemented. Further extrapolation of these results are discussed in section *5.1 Findings*.

Chapter 5

Conclusion

With the continued improvement rate of computer speed, memory, and miniaturization postulated by Moore's Law, image based feedback may become ubiquitous among robotic vehicles. Image feedback offers distinct advantages compared with many state of the art local navigation proximity systems. Wide viewing angle, low cost, and platform growth potential make cameras an extraordinarily versatile and capable sensor system. Additionally, the control strategies implemented by this paper are intuitive, simple and successful; the control law stably and reliably guides the vehicle along a corridor or away from looming objects. Concerns regarding computational demand, noise susceptibility and real-time capability proved to be unfounded as is clearly indicated by the successes and rates demonstrated by the implementation.

5.1 Findings

Indoor and outdoor flights all exceeded expectations and demonstrated controller resilience in the presence of environmental variations. The final implementation exhibited systematic successes transiting the corridor and avoiding obstacles looming in the periphery. The controller showed minor weakness when focused on or near feature poor regions. The current implementation does not adequately handle feature poor areas; in particular, scenarios where one of the left or right panels has no features. Experience suggests that feature poor regions are commonly associated with proximate objects—distant, feature-poor objects are generally able to acquire a keypoint

here or there—and therefore, featureless regions could simply be treated as if they were closer than feature dense regions. Determining the action in this situation is slightly more nuanced than first glance suggests. The problem of adapting to feature poor environments should be researched in further detail. The problem shares roots with Gestalt theory; differentiating between objects and their negative spaces.

A number of significant accomplishments were realized throughout research and implementation. Feature point introduction using FAST and FReaK offer critical speed-ups enabling real time, robust keypoint tracking. The FAST grid adaptation allows for targeted keypoint introduction into feature poor regions. Other detectors and descriptors, namely the competing BRISK, performed similarly well to FAST and FReaK for low volume point introduction, however, failed to produce keypoints as robust as FReaK or in as great quantities as the FAST-FReaK combination. Testing also confirmed that coupling sparse feature detection with the Lucas-Kanade sparse optical flow algorithm demonstrated high rates, reliable optical flow and the ability to tune feature persistence against drift error. Two powerful filtering methods capably select unique and robust feature points: Lowe matching ratio and back calculated optical flow. The tuning of back calculated optical flow to control keypoint persistence or accuracy is believed to be an original contribution of this thesis.

The control implementation demonstrates that time-to-transit is a viable feedback signal. The testing offered strong empirical evidence supporting τ balancing as a valid control strategy. The control law makes assumptions which limit the stability region to near planar and near linear motions. These assumptions may be relaxed with rotational field rejection and active determination of the focus of expansion. Accounting for noise and signals induced by rotational motions will improve controller performance, but more importantly resolve issues allowing the removal of restrictive assumptions. This would enable a larger range of working motions. Still,

the controller as presented adequately handles camera skew from the direction of motion and minor rotational variations. Moreover, the averaging strategy inherently mitigates positive feedback loops associated with rotational motions. The ability to overcome the Larsen effect stems from the symmetry and uniformity of rotational fields coupled with the averaging strategy. The control signal is produced by adding or subtracting right and left panel loom to produce the smallest resulting signal. Because rotationally induced flow components are generally equal in the left and right panels the addition or subtraction performed when calculating the signal very nearly cancels the rotational components.

Despite initial concerns and a number of attempts to reduce latency between the vehicle and computer system, system latency was circumvented by reducing the vehicle speed, increasing controller response rate relative to system time constants. Latency should be considered when producing the next test vehicle to eliminate vehicle speed restrictions.

The current image processing library performs all calculations faster than the frame rate of the camera. These parameters may be tuned to achieve different feedback qualities. The key features which may be leveraged in order to optimize processing speed are: Image size ($w \times h$), number of features to introduce, LK-optical flow pyramid depth and neighborhood size and descriptor matching method (L^1 norm vs. L^2 norm vs. Hamming distance).

5.2 Future Work

The accomplishments and successes demonstrated by using time-to-transit as a feedback signal offer strong reason for further exploration and research into this subject. The field is rich with exciting theory and ample opportunity for original work. At the time of writing, a *New York Times* article titled *'Now, Anyone Can Buy a Drone.*

Heaven Help Us' written by Nick Wingfield explores the recent boom of personal drones. In particular, it reviews how lack of an administering body has created a void which has been filled by innovations and abuses. The article tone suggests annoyance verging on amusement and implies how, to many, drones are quite simply a public nuisance. The article concludes fatedly that despite the assured regulation of drones, the technology is too capable and desirable to be stifled; drone development will be spirited, swift and unfettered.

While creating the implementation presented by this thesis, the vast quantity of interconnecting fields provided ample opportunity for project scope creep. Initial exploration down a number of these avenues successfully (and excitingly) delayed thesis completion by as many as three months. Even now, other interesting doors remained unopened. Further research should be performed to improve feature introduction and tracking performance, reduce high gain error, increase operating region and region of stability, develop versatile control mechanisms and upgrade the vehicle platform.

Current feature point introductions occur randomly throughout the image, focusing on robustness over location preference. Work was initiated to allow selective introduction of points into low marker regions. By dispersing keypoints throughout the frame, collisions caused by negligence—overlooking objects—become less likely.

Keypoint tracking using LK-optical-flow accounts for the highest processing time of all computations. Time delays are augmented when points are both tracked and introduced during the same iteration. After performing matching during the point introduction phase, optical flow information is available for the small subset of newly introduced points. The flow information could be leveraged to reduce the size of the Lucas-Kanade search window and speed up the optical flow calculations for that iteration.

The current implementation tracks points and maintains records for an arbitrary

number of frames, j. This tracking has been useful for visualizing flow fields, but other than this, was not leveraged for control or error reductions. Keypoint temporal information is highly valuable and could be leveraged to anticipate future motions. One could couple vehicle system dynamics with a spatial reduction constraint and some form of numerical differential equation estimator (Runga-Kuta or Adams-Bashforth) to determine anticipated motions. Using such a system could change the face of this algorithm completely, ranging from faster and error free optical flow calculations to predictive avoidance systems.

While adequately managed in the control, dissecting rotationally induced flow fields from translational ones could drastically reduce noise and improve the control. In the same vein, calculating the focus of expansion would provide the vehicle with direct information about its heading and more accurate predictions for time-to-transit. Both issues are coupled and remain significant challenges in image processing. Research in the field suggests there are tools available to handle such problems. One particularly good paper was presented by Royden in [29] and should serve as an introduction to the topic. The algorithm proposed is slightly computationally demanding, but with clever implementation could be reduced into a usable form.

Attempts at calculating the focus of expansion failed in almost all regards. While the focus would occasionally appear in the correct location, the lack of reliability made the calculation less than ideal for control. The focus would often jump off the image, or around the image, even after employing a moving average strategy. It is believed that the jumping may be caused performing the incorrect decomposition of the homography matrix (there are four such decompositions).

The current control implementation is powerful for its simplicity. Still, with minor tweaks, the system could enable higher order controls to circle objects or move around corners. Some examples of these controls are demonstrated in [1]. One key

requirement of these controllers is the ability to track objects. Tracking a single keypoint or maximizing the difference, τ, between two keypoints is risky given how often points enter, exit or drop from the frame. Alternatively, identifying points associated with a unique object—let alone identifying whether a given object is significant—is incredibly challenging. Initial steps were taken towards creating a simple object tracking system. The image is segmented into an $n \times n$ grid and the average value of *tau* determined for each region. The averages are used to determine variance for each keypoint relative to the cell in which it resides. Next, the minimal enclosing circle is calculated to encompass all points of a given variance. Significance for the region is determined by weighing the significant (non-variant points) against the quantity of outliers (variant points) in the region. This method acceptably locates the centroid of whichever object is driving *tau* within a given cell.

Detailed review of flight videos demonstrated that the simple object focusing strategy consistently identified proximal looming objects. One minor flaw was in regions containing two objects on similar distal planes, where the object center would be defined in the region between the two objects rather than as two separate looming objects. The object tracking strategy was tested successfully for grids as large as a 12×12 yet dimensioning returns (tracking insignificant objects) were observed for grid sizes as low as 5×5. Higher order grids were found to slow software calculations considerably.

The current implementation uses a vehicle with a forward facing camera. As reviewed in [1], there are numerous advantages for using a side facing camera. Some advantages are reviewed near the end of *section 3.2.2 Control Overview*. The side viewing camera emphasizes the center of the image for transited objects, taking the focus from collision avoidance and re-centering it on path guidance.

One final topic in the field of improved controllability would be additional research

into image segmentation. The current tryptic segmentation strategy offers a simple and intuitive solution to the flow balancing problem. Still, minor tweaks could not only balance the vehicle between horizontally approaching obstacles but also vertical ones. Furthermore, the center panel could be tested for looming objects and the vehicle instructed to avoid collisions. With minor research, segmentation strategies might be developed into more robust and capable control laws.

The final recommendation for future work would be upgrading or improving the robotic platform. The AR.Drone2 was a boon to the project, primarily allowing for rapid prototyping and testing of control algorithms. Switching to a system with on board processing would certainly sacrifice the ease of use associated with the Ar.Drone2. The benefits would be eliminating all communications delays within the internal control loop. Large effort was invested attempting to correct the latency issues between the computer and AR.Drone2. The issue was much more persistent than originally expected. Latency on board the vehicle remains an open challenge in moving forward with the project.

Appendix

```
"""
Image Processing Library

Provides a number of image processing systems to introduce and track keypoints
in an image.
"""
__author__ = 'Paul Seebacher'

# Python Standard libraries
import time
from collections import deque
import copy
import warnings
import csv

# Image processing and numerical calculations
import cv2
import numpy as np

# Generic set up and selection of detector, descriptor and matcher objects
DETECTOR_TYPES = {1: "FAST", 2: "GridFAST", 3: "BRISK"}
DESCRIPTOR_TYPES = {1: "FREAK", 2: "BRISK"}
MATCHER_TYPES = {1: "BRUTEFORCE"}

DETECTOR = DETECTOR_TYPES[2]
DESCRIPTOR = DESCRIPTOR_TYPES[1]
MATCHER = MATCHER_TYPES[1]

MTCH_METHODS = [cv2.NORM_L1, cv2.NORM_L2, cv2.NORM_HAMMING]

MTCH_METHOD = MTCH_METHODS[2]    # HAMMING for binary descriptors
MATCH_QTY = 1500                 # number keypoints desired per image

# Parameters for Lucas-Kanade optical flow
LK_PARAMS = dict(winSize=(10, 10),
                 maxLevel=2,
                 criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
                           10, 0.03))

# This parameter adjusts resilience of backwards calculated points from LK
# optical flow. The higher the number, the more a point will persist from
```

```
# lucas kanade. Basically, this means points stick around on objects with
# large motions or have a higher chance to reattach to different pixels.
# The downside is the higher this number is, the more likely flow vectors
# will be incorrect under large motions. Better to keep it low (20 max) and
# introduce new points rather than retain old, often incorrect ones.
LK_BACKCALC_PIXEL_DEVIATION = 25

# Number of (x,y) points retained for keypoint track history
MAX_TRACK_LENGTH = 5


class Image(object):
    """
    Base container class used in the image processing library
    """
    def __init__(self, frame, frame_time=None):
        height = frame.shape[0]
        width = frame.shape[1]
        size = (width, height)

        self.size = size
        if frame_time is None:
            self.frame_time = time.time()
        else:
            self.frame_time = frame_time
        self.clr_img = frame
        self.gry_img = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Lazily set attributes. Test for existence before use
        self.kp = []
        self.desc = []
        self.tracks = []

    def copy(self):
        return copy.deepcopy(self)

    def get_color(self):
        return self.clr_img

    def get_grey(self):
        return self.gry_img
```

```
    def get_time(self):
        return self.frame_time

    def get_size(self):
        return self.size

    def set_details(self, kp, desc):
        self.kp = kp
        self.desc = desc

    def set_flow_tracks(self, tracks):
        self.tracks = tracks

    def get_kp(self):
        return self.kp

    def get_desc(self):
        return self.desc

    def get_flow_tracks(self):
        return self.tracks


class ImageCompare(object):
    def __init__(self):
        """
        Enacts a number of comparison techniques to track key points between
        frames from a video source or individual pictures.

        Automatically tracks time and retains a history of previous image data.

        Class loads ImageData instances with single instance of OpenCV
        descriptor and detector classes.
        """

        # Data attributes storing frame data
        self.frame_prev = None
        self.frame_curr = None

        # Data attributes storing comparison data
        self.matches_curr = None
```

```
        # creates detector descriptor and matcher
        self.detector = cv2.FeatureDetector_create(DETECTOR)
        if "grid" in DETECTOR.lower():
            grid_adapted = cv2.GridAdaptedFeatureDetector
            self.detector = grid_adapted(self.detector, MATCH_QTY/3, 6, 6)
        self.descriptor = cv2.DescriptorExtractor_create(DESCRIPTOR)

        if MATCHER == MATCHER_TYPES[1]:   # BruteForce
            self.matcher = cv2.BFMatcher(MTCH_METHOD, crossCheck=False)

        # Data attributes storing time tracking
        self.t0 = None
        self.t_prev = None
        self.t_curr = None
        self.dt = None

        # previous points and tracked points (list of dequoues)
        self.points = []
        self.tracks = []

    def compare(self, image):
        """
        Action: Compares two images using selected method
        Input:  Image instance
        Output: (Current keypoints, Previous Keypoints)
        """

        # Advance time tracking with new image time
        frame_time = image.get_time()
        self.__track_time(frame_time)

        # Advance state machine image storage
        self.frame_prev = self.frame_curr
        self.frame_curr = image

        # Calculate optical flow for keypoints. Determine new from old position
        new_points, old_points = self.__optical_flow(tracking=True)

        return new_points, old_points

    def get_dt(self):
        """
```

```
        Action: Returns delta t between two frames
        Input: None
        Output: time difference between current and previous frame
        """
        return self.dt

    def __detector_descriptor(self, image, mask=None):
        """
        Action: Add key points and key point descriptors to Image instance
        Input:  Image object being tested
                optional mask of areas to test (same shape as frame)
        Output: Updated image object with keypoints and descriptors
        """

        # Get grey scale frame and create mask of ones if nothing passed in.
        frame_grey = image.get_grey()
        mask = np.ones_like(frame_grey) if mask is None else mask

        # Detect keypoints and time function call
        TIMER.set()
        kp = self.detector.detect(frame_grey, mask=mask)
        TIMER.delta("Detector")

        # Create descriptors for keypoints and time function call
        TIMER.set()
        (kp, desc) = self.descriptor.compute(frame_grey, kp)
        TIMER.delta("Descriptor")

        # Lazily set keypoint and descriptors on Image object
        image.set_details(kp, desc)

        return image

    def __match(self):
        """
        Action: Determine matches between two frames
        Input: None
        Output: List of match objects (each object has index of train and query
                descriptors. Use to create list of matching features)
        Note: Call this function after using "__detector_descriptor" method
        """
```

```
        # Access descriptors from current and previous frame.
        # Need descriptors from both images to create matches.
        desc_curr = self.frame_curr.get_desc()
        desc_prev = self.frame_prev.get_desc()

        # If there are no descriptors associated with previous frame default
        # is to return empty list (meaning no matches)
        if desc_prev is None:
            return []

        if len(desc_prev) == 0:
            return []

        # Try to perform knn (k-nearest-neighbors) match. Return two closest
        # matches for keypoint filtering by Lowe ratio test
        TIMER.set()
        knn_kwargs = {'queryDescriptors': desc_curr,
                      'trainDescriptors': desc_prev,
                      'k': 2}
        try:
            raw_matches = self.matcher.knnMatch(**knn_kwargs)
        except cv2.error:
            raw_matches = []
        else:
            TIMER.delta("Matching")

        # Perform Lowe-ratio test to discard 90% of false matches and reject
        # less than 5% of correct matches
        matches = self.__lowe_filtering(raw_matches)

        return matches

    def __optical_flow(self, tracking=False):
        """
        Action: Perform point introductions if fewer keypoints than threshold
                Track keypoints with Lucas-Kanade optical flow w/ filtering
        Input: Optional flag for tracking: retain point correspondences for
               MAX_TRACK_LENGTH number of frames
        Output: returned_new_keypoints - current frame keypoints
                returned_old_keypoints - previous frame keypoints
        """
```

```
# global constants which have largest impact on processing time
global MAX_TRACK_LENGTH
global MATCH_QTY
global LK_BACKCALC_PIXEL_DEVIATION

# Frame margin ignored to prevent index out of bounds from descriptors.
EDGE_MARGIN = 5

# Default is to return empty list of points
returned_new_points = []
returned_old_points = []

# if compare function has only been called once, no frame to compare to
if not self.frame_prev:
    return returned_new_points, returned_old_points

points = self.points
points_detected = False
# If fewer points than threshold, introduce new keypoints
# NOTE: If entered, this section may modify points before performing
#       Optical Flow calc under next conditional.
# NOTE: This section may be entered knowing points will not be matched
#       and points NOT introduced as the descriptors from the current
#       frame need to be readied for the following frame.
if len(points) < MATCH_QTY:
    # Perform detection, description and matching
    self.frame_curr = self.__detector_descriptor(self.frame_curr)
    matches = self.__match()
    points_detected = True

    TIMER.set()

    # The remainder of the point introduction block concatenates the
    # newly discovered keypoints onto the previous set of points.
    #
    #           *** This is actually slightly tricky ***
    #
    # Because the previous points are all defined relative to the train
    # image we MUST use the keypoints from the train image rather than
    # query image. (despite the query being newer). This way, when the
    # LK-OptFlow algorithm is executed in the next block, there will be
    # agreement between the carried over points which associate with
```

```
        # the previous image. If this isn't done and new points are used,
        # the (x,y) positions which associate with the new frame will be
        # applied to the previous frame and will almost certainly be
        # different keypoints. This means the points will not be as robust
        # as those we've just worked so hard to acquire.
        kp_prev = self.frame_prev.get_kp()

        # Perform indexing to create list of (x,y) coords for best matches
        best_kp_prev = [kp_prev[m.trainIdx].pt for m in matches]

        # If tracking, create new deques for newly introduced points
        if tracking:
            for x, y in best_kp_prev:
                self.tracks.append(deque([(x, y)],maxlen=MAX_TRACK_LENGTH))

        # Optical flow algorithm executes next using "points" variable.
        # Conditional makes sure optical flow gets newly introduced AND
        # retained points both associated with the older frame.
        # Also deals with edge cases when no points are introduced.
        #
        # If no old points or new points, points is an empty list
        if len(points) == 0 and len(best_kp_prev) == 0:
            points = []
        # If no old points, just float and return newest points
        elif len(points) == 0:
            points = np.float32(best_kp_prev)
        # If no new points are found, return previous points
        elif len(best_kp_prev) == 0:
            pass
        # Otherwise, concatenate new and old points and return
        else:
            points = np.concatenate((points, np.float32(best_kp_prev)), 0)

        returned_new_points = points
        TIMER.delta("Point Insertion")

    # Perform optical flow calculations if there are keypoints
    if len(points) > 0:
        # Store references to frame size, current and previous grey scales
        width, height = self.frame_curr.get_size()
        grey_curr = self.frame_curr.get_grey()
        grey_prev = self.frame_prev.get_grey()
```

```
# Renamed to shorten most function calls
kp_0 = points

# Perform optical flow calculations
TIMER.set()
lk_args = [grey_prev, grey_curr, kp_0, None]
kp_1, sts, flg = cv2.calcOpticalFlowPyrLK(*lk_args, **LK_PARAMS)
TIMER.delta("LK-Flow")

new_points = kp_1
old_points = kp_0

# Filtering steps
# Knock off all keypoints within the edge margin. This stops index
# errors for descriptors outside or on frame margin. Also, this
# means for moving frame, points moving 'off' the frame are dropped
# so new points may be introduced
TIMER.set()
x_good = (new_points[:, 0] > EDGE_MARGIN) & \
         (new_points[:, 0] < width - EDGE_MARGIN)

y_good = (new_points[:, 1] > EDGE_MARGIN) & \
         (new_points[:, 1] < height - EDGE_MARGIN)

# Performing filtering by running LK-OptFlow ::backwards:: meaning,
# use the query image as the train image and train image as the
# query image. Only take points that match in both directions.
lk_args = [grey_curr, grey_prev, kp_1, None]
kp_0r, sts, flg = cv2.calcOpticalFlowPyrLK(*lk_args, **LK_PARAMS)
pxl_delta = abs(kp_0-kp_0r).max(1)

r_good = pxl_delta < LK_BACKCALC_PIXEL_DEVIATION
TIMER.delta("LK-Back-Filter")

TIMER.set()
# Create a mask which is True for points that are not in margins,
# and meet filtering requirements
good = r_good & x_good & y_good

# Perform indexing to select high quality keypoints
indices = [i for (i, flag) in enumerate(good) if flag]
```

```
            filtered_new_points = new_points[indices, :]
            filtered_old_points = old_points[indices, :]

            # drop tracks whose points have not persisted and update detected
            if tracking:
                good_tracks = zip(good, self.tracks)
                new_tracks = [track for (flag, track) in good_tracks if flag]
                for point, new_track in zip(filtered_new_points, new_tracks):
                    new_track.append(point)
                self.tracks = new_tracks

            returned_new_points = filtered_new_points
            returned_old_points = filtered_old_points
            TIMER.delta("Track-Updates")

            # Code optimization
            # Run detector and descriptor if filtering has caused keypoints to
            # fall below threshold. This way, detection and description are
            # performed preemptively so new points may immediately be
            # introduced in the next iteration. Preferable than calling
            # detection-description twice on next iteration as this causes
            # calculation speeds to fall outside of 30 hz region
            if len(filtered_new_points) < MATCH_QTY and not points_detected:
                self.frame_curr = self.__detector_descriptor(self.frame_curr)

        self.points = returned_new_points

        # New points and old points should always match in length as they
        # should correspond. Simple test that may show errors with changes
        # to code.
        assert len(returned_new_points) == len(returned_old_points)

        return returned_new_points, returned_old_points

    @staticmethod
    def __lowe_filtering(raw_matches):
        # Performs the Lowe ratio test on a matched set of keypoints
        TIMER.set()

        # Match tolerance ranges
        low_tol = 0
```

```
        hgh_tol = 10
        accepted = 6  # and lower

        # buckets matches based on ratio
        match_quality = [[] for _ in xrange(low_tol, hgh_tol+1)]
        try:
            for (m, n) in raw_matches:
                try:
                    bucket = int(10*m.distance/n.distance)
                except ZeroDivisionError as err:
                    # two matches on top of one another. Rare but acceptable
                    bucket = 0
                if bucket < hgh_tol:
                    match_quality[bucket].append(m)
        except ValueError:
            match_quality[hgh_tol].extend([match[0] for match in raw_matches])

        matches = []

        num_matches = 0

        # Select best matches based on their ratio distance
        for i in xrange(accepted):
            bckt_qty = len(match_quality[i])
            if num_matches > MATCH_QTY:
                break
            elif bckt_qty < MATCH_QTY - num_matches:
                matches.extend(match_quality[i])
                num_matches += bckt_qty
            else:
                matches.extend(match_quality[i][:MATCH_QTY - num_matches])
                num_matches += MATCH_QTY - num_matches + 1

        TIMER.delta("Lowe-Filtering")

        return matches

    def __track_time(self, frame_time=None):
        """
        Private method for tracking dt, current and previous time steps
        relative to an initial starting point
        """
```

```
        # Store initial time
        if self.t0 is None:
            self.t0 = frame_time

        t_prev = self.t_curr
        t_curr = frame_time

        # frames are assumed to be sequential in time. Error if dt is negative
        if t_prev is not None:
            assert t_curr - t_prev >= 0
            self.dt = t_curr - t_prev

        self.t_curr = t_curr
        self.t_prev = t_prev

    def create_flow_plot(self, kp_0=None, kp_1=None):
        """
        Action: Creates visualization of keypoint motion between frames. To
                include color coding of keypoints based on optical flow rate,
                kp_0, and kp_1 must be provided: the location of the previous
                and current keypoints.

        Input: Optional - kp_0 - previous keypoint positions
                          kp_1 - current keypoint positions
        Output: Image object built from current frame, keypoints and keypoint
                tracks.
        Class Requirements: Current color image object
                            Current images Keypoints
                            (Uses) Keypoint track history when available.
        """
        colr_curr = self.frame_curr.get_color()
        frame_time = self.frame_curr.get_time()

        TIMER.set()

        # If user has passed previous and current points in use rate of motion
        # to "color code" tracks based on rate of motion
        if kp_0 not in [None, []] and kp_1 not in [None, []]:
            dxy = kp_1 - kp_0
            dist = np.linalg.norm(dxy, axis=1)
            rgb_vals = scalarMap.to_rgba(dist, bytes=True)[:, 0:3].astype(int)
```

```
            rgb_vals = map(tuple, rgb_vals)
        # Otherwise just set all the track colors equally. This is faster.
        else:
            def color_iter():
                while True:
                    yield (0, 255, 0)
            rgb_vals = color_iter()

        # plot the current points using a circle of the appropriate color.
        for pt, color in zip(self.points, rgb_vals):
            x, y = pt
            cv2.circle(colr_curr, (x, y), 2, color, 1)

        # if tracking is turned on, plot the tracks with appropriate color.
        if self.tracks is not []:
            for track, color in zip(self.tracks, rgb_vals):
                cv2.polylines(colr_curr, [np.int32(track)], False, color)

        # Plot the number of points being tracked at the top of the image.
        draw_str(colr_curr, (20, 20), 'track count: %d' % len(self.points))
        TIMER.delta("Plotting-Tracks")

        # Return the new image for display.
        colr_curr = Image(colr_curr, frame_time=frame_time)
        return colr_curr

    def create_grid_plot(self, cell_mags, (num_rows, num_cols), max_norm=10.0):
        """
        Action: Creates a bar plot in each cell of an m x n grid. Bar plot will
                be x and y directions.
        Input: cell_mags - list of x-y magnitude [(x1, y1),(x2,y2),...,(xn,yn)]
               num_rows  - number of rows bar plots will be applied to.
               num_cols  - number of cols bar plots will be applied to.
        Class Requirements: Current color image object.
        """
        colr_curr = self.frame_curr.get_color()
        frame_time = self.frame_curr.get_time()

        if cell_mags == [] or cell_mags == None:
            return self.frame_curr

        TIMER.set()
```

```
width, height = self.frame_curr.get_size()
cols = np.linspace(0, width, num_cols+1)
rows = np.linspace(0, height, num_rows+1)

col_width = (cols[1]-cols[0])
row_width = (rows[1]-rows[0])
col_centers = cols[1:] - col_width/2.0
row_centers = rows[1:] - row_width/2.0

rect_half_width = 5
rect_max_length = width/num_cols
rect_max_height = height/num_rows

for i, mags in enumerate(cell_mags):
    col_indx = i % num_cols
    row_indx = i / num_cols
    cent_xy = (col_centers[col_indx], row_centers[row_indx])

    # x-directional rectangle
    if np.isnan(mags[0]):
        pass
    else:

        x1 = int(cent_xy[0])
        y1 = int(cent_xy[1] - rect_half_width)
        x2 = int(cent_xy[0] + mags[0]/max_norm*rect_max_length)
        y2 = int(cent_xy[1] + rect_half_width)
        cv2.rectangle(colr_curr, (x1, y1), (x2, y2), (0, 255, 0), -1)

    # y-directional rectangle
    if np.isnan(mags[1]):
        pass
    else:
        x1 = int(cent_xy[0] - rect_half_width)
        y1 = int(cent_xy[1])
        x2 = int(cent_xy[0] + rect_half_width)
        y2 = int(cent_xy[1] + mags[1]/max_norm*rect_max_height)
        cv2.rectangle(colr_curr, (x1, y1), (x2, y2), (0, 0, 255), -1)

TIMER.delta("Plotting-Grid")
return Image(colr_curr, frame_time=frame_time)
```

```
def control_calculations(new_pts, old_pts, dt, frame):
    """
    Action: Performs calculations to determine time-to-transit and use the
            the values to create a control signal from the video feed.
            Additional features such as FOE calculations and object association
            calculations are also performed here. These features may be be
            developed or removed as desired.
    Input:  new_pts - keypoints locations from current frame
            old_pts - keypoints locations from previous frame.
            dt      - time difference between two frames
            frame   - image object for current frame (used for plotting)
    Output: control_image - Image with control signal plotted on it
            control       - Control signal
            control_style - Form of control (divergent or uniform flow)
            foe_cent      - Attempt at location of focus of expansion

    """
    global COLORS
    _ = (None, None, None)

    if new_pts == [] or old_pts == []:
        return frame, _
    if not new_pts.size or not old_pts.size:
        return frame, _
    if frame is None:
        return frame, _

    TIMER.set()
    width, height = frame.get_size()
    x_cent = width/2
    y_cent = height/2

    # Cartesian
    dxy = new_pts - old_pts
    dxy_dt = dxy/dt
    xy_pos = new_pts - (x_cent, y_cent)
    loom = dxy_dt/xy_pos

    # Focus of Expansion
    dyx_dt = np.fliplr(dxy_dt)
    b = -np.diff(new_pts * dyx_dt)
```

```
try:
    foe = np.linalg.inv(dyx_dt.T.dot(dyx_dt)).dot(dyx_dt.T).dot(b)
    foe_cent = (int(foe[0]), abs(int(foe[1])))
except ValueError:
    foe_cent = (width/2.0, height/2.0)
except np.linalg.LinAlgError:
    foe_cent = (width/2.0, height/2.0)

# Cylindrical
r = np.linalg.norm(xy_pos, axis=1)
dr_dt = np.sum(xy_pos*dxy_dt, axis=1)/r
#tht = np.arctan2(xy_pos[:, 1], xy_pos[:, 0])
#dtht_dt = np.diff(xy_pos*dxy_dt[:, [1, 0]], axis=1)/r

# Time-to-transit
tau = r/dr_dt
TIMER.delta("Tau-Calculations")

# Field Indexing
TIMER.set()
rows = 1
cols = 3
bounds = ((0, width), (0, height))
bin_numbers = bin_indexing2d(new_pts, rows, cols, bounds)
TIMER.delta("Histogram-Binning")

TIMER.set()
# Filtering
tau_threshold = 10
tau_emergency = 1
saliency_cutoff = width/4.0

tau_threshold_mask = np.absolute(tau) < tau_threshold
tau_filtered = tau[tau_threshold_mask]
bin_filtered = bin_numbers[tau_threshold_mask]
pos_filtered = new_pts[tau_threshold_mask]

tau_emergency_mask = np.absolute(tau) < tau_emergency
tau_crit_pts = new_pts[tau_emergency_mask]
tau_crit_vls = tau[tau_emergency_mask]

bin_indices = [bin_filtered == indx for indx in xrange(1, rows*cols+1)]
```

```
# Calculate quantity, average and stdev for tau samples in each grid cell
cell_taus_qty = [np.sum(bin_index) for bin_index in bin_indices]
cell_taus_avg = [np.average(tau_filtered[bin_index], axis=0)
                 for bin_index in bin_indices]
cell_taus_std = [np.std(tau_filtered[bin_index], axis=0, ddof=1)
                 for bin_index in bin_indices]

# Create a list of lists storing the indices of points which are within
# +/-1 sigma of tau within a given cell. There will be m x n such groups
# with an unknown number of points associated with the group.
group_indices = [(tau_filtered > cell_taus_avg[i]-cell_taus_std[i]) &
                 (tau_filtered < cell_taus_avg[i]+cell_taus_std[i]) &
                 bin_index
                 for i, bin_index in enumerate(bin_indices)]

# Using the index marking points with tau within +/-1 sigma within a given
# cell, create the list of m x n groups of points from the index so that
# those points may be acted on quickly in future calculations.
group_points = [pos_filtered[group_indx] for group_indx in group_indices]

# Calculate the number of points associated for each grouping of points
group_xy_qty = [group.shape[0] for group in group_points]

# Calculate the mean position of each grouping of points
group_xy_avg = [np.mean(group, axis=0) for group in group_points]

# Calculate the scatter (x-y positional deviation) within a given grouping
# of points
group_xy_std = [np.std(group, axis=0, ddof=1) for group in group_points]

# Run through a similar process on each group to down select previously
# grouped points by chosing those within +/-2 sigma on positional variation
# In other words, select a tight group of the previously selected points.
# The following lines create the index of such points, followed by a list
# containing the groups of such points (from the index).
indx_2sigma = [(group > (group_xy_avg[i]-2*group_xy_std[i])) &
               (group < (group_xy_avg[i]+2*group_xy_std[i]))
               for i, group in enumerate(group_points)]
group_2_sig = [group2s[:, 0] & group2s[:, 1] for group2s in indx_2sigma]

# Determine the centroids and radii for each group. This is the proposed
```

```
# "object" determined by this algorithm.
group_center = [np.average(points[group2s], axis=0)
                for group2s, points in zip(group_2_sig, group_points)]
iterable = enumerate(zip(group_2_sig, group_points))
group_radius = [np.linalg.norm(points[group2s] - group_center[i], axis=1)
                for i, (group2s, points) in iterable]
group_radii = []
for group_rad in group_radius:
    try:
        group_radii.append(group_rad.max())
    except ValueError:
        group_radii.append(0)

colr_curr = frame.get_color()
frame_time = frame.get_time()

# Plot the objects determined by the algorithm
for (x, y), r, colr in zip(group_center, group_radii, COLORS):
    try:
        cv2.circle(colr_curr, (x, y), 10, colr, 1)
    except ValueError:
        pass
TIMER.delta("Tau-object-grouping")

TIMER.set()

# Control signal created here
if np.sign(cell_taus_avg[0]) != np.sign(cell_taus_avg[2]):
    if cell_taus_avg[0] < 0:    # leftward moving vehicle
        motion = -1
    else:                       # rightward moving vehicle
        motion = 1
    control = motion*(1.0/cell_taus_avg[0] + 1.0/cell_taus_avg[2])
    control_style = "uniform"
else:
    control = (1.0/cell_taus_avg[0] - 1.0/cell_taus_avg[2])
    control_style = "divergent"

# Plot the loom signal on the left side of the image
try:
    left_center = (width/4, height/2 - 5)
    left_corner = (int(width/4*1.0/cell_taus_avg[0])+width/4, height/2+5)
```

```
            left_pt_a = (width/4 - 2, height/2 - 2)
            left_pt_b = (width/4 + 2, height/2 + 2)
        except ValueError:
            pass
        else:
            cv2.rectangle(colr_curr, left_center, left_corner, (0, 0, 255), -1)
            cv2.rectangle(colr_curr, left_pt_a, left_pt_b, (0, 0, 255), -1)

        # Plot the loom signal on the right side of the image
        try:
            right_center = (width*3/4, height/2 - 5)
            right_corner = (width*3/4-int(width/4*1./cell_taus_avg[2]), height/2+5)
            right_pt_a = (width*3/4 - 2, height/2 - 2)
            right_pt_b = (width*3/4 + 2, height/2 + 2)
        except ValueError:
            pass
        else:
            cv2.rectangle(colr_curr, right_center, right_corner, (0, 0, 255), -1)
            cv2.rectangle(colr_curr, right_pt_a, right_pt_b, (0, 0, 255), -1)

        # Plot the control signal in the center of the image
        try:
            center_root = (width/2, height/2-10)
            center_corner = (int(width/4*control)+width*2/4, height/2+10)
        except ValueError:
            pass
        else:
            cv2.rectangle(colr_curr, center_root, center_corner, (0, 255, 0), -1)

        control_image = Image(colr_curr, frame_time=frame_time)
        TIMER.delta("Signal_plotting")

        return control_image, (control, control_style, foe_cent)


def bin_indexing2d(xy_data, m_rows=2, n_cols=2, xy_window=((0, 1), (0, 1))):
    """
    For m rows and n columns for the i-th row and j-th column the general form
    bin zones indices are defined in the xy-plane as

                        Y
                        ^
```

```
        xy_window[1][1]_|
                        |  |  1  |  2  |     ...     |  n  |
                        |  | n+1 | n+2 |     ...     | ... |
                        |  | ... | ... | (i-1)*n + j | ... |
        xy_window[1][0]_|  | ... | ... |     ...     | m*n |
                        |
                        |--|---------------------------------|---> X
                          xy_window[0][0]           xy_window[0][1]

    """
    m_rows = int(m_rows)
    n_cols = int(n_cols)

    if m_rows < 1:
        raise IOError("Need at least 1 row")
    elif n_cols < 1:
        raise IOError("Need at least 1 column")

    if not xy_data.size:
        warnings.warn("returning empty list may cause errors")
        return []

    # Each index ``i`` returned is such that ``bins[i-1] <= x < bins[i]`` if
    # correct for fence post issue
    cols = np.linspace(xy_window[0][0], xy_window[0][1], n_cols+1)
    rows = np.linspace(xy_window[1][0], xy_window[1][1], m_rows+1)

    x_data = xy_data[:, 0]
    y_data = xy_data[:, 1]

    x_col_index = np.digitize(x_data, cols)
    y_row_index = np.digitize(y_data, rows)

    zone_index = (y_row_index-1)*n_cols + x_col_index
    return zone_index


# Class for debugging and timing various processes
class DebugTimer(object):
    def __init__(self, debug_on=True):
        self.set_time = time.time()
        self.delta_time = {}
```

```
        self.delta_record = {}
        self.order = []
        self.debug_on = debug_on

    def __str__(self):
        if not self.debug_on:
            return ""

        total = 0.0
        tostr = ""
        for block in self.order:
            tostr += "%7.5f - %s\n" % (self.delta_time[block], block)
            total += self.delta_time[block]
            self.delta_time[block] = 0
        tostr += "%6.4f - %s" % (total, "Total Processing Time")
        return tostr

    def set(self):
        if not self.debug_on:
            return

        self.set_time = time.time()

    def delta(self, block_name):
        if not self.debug_on:
            return

        dt = time.time() - self.set_time
        if block_name not in self.delta_time:
            self.order.append(block_name)
        self.delta_time[block_name] = dt
        try:
            record = self.delta_record[block_name]
        except KeyError:
            self.delta_record[block_name] = []
            record = self.delta_record[block_name]
        record.append(dt)

    def save_record(self, name):
        record = self.delta_record
        with open(name, "wb") as outfile:
            writer = csv.writer(outfile)
```

```
            writer.writerow(record.keys())
            writer.writerows(zip(*record.values()))

TIMER = DebugTimer()


def draw_str(dst, (x, y), s):
    blk = (0, 0, 0)
    wht = (255, 255, 255)
    cv2.putText(dst, s, (x, y), cv2.FONT_HERSHEY_PLAIN, 1.0, blk, thickness=2)
    cv2.putText(dst, s, (x+1, y+1), cv2.FONT_HERSHEY_PLAIN, 1.0, wht)

# Various features for plotting
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.cm as cmx

import colorsys

jet = plt.get_cmap('jet_r')
cNorm = colors.Normalize(vmin=0, vmax=25)
scalarMap = cmx.ScalarMappable(norm=cNorm, cmap=jet)


def get_colors(num_colors, normed=False):
    colorlist=[]
    for i in np.arange(0., 360., 360. / num_colors):
        hue = i/360.
        lightness = (50 + np.random.rand() * 10)/100.
        saturation = (90 + np.random.rand() * 10)/100.
        rgb_norm = colorsys.hls_to_rgb(hue, lightness, saturation)
        rgb_255 = tuple(map(lambda x: int(255*x), rgb_norm))

        if not normed:
            colorlist.append(rgb_255)
        else:
            colorlist.append(rgb_norm)

    return colorlist

COLORS = get_colors(4)
```

Bibliography

[1] Z. Kong, K. Ozcimder, N. Fuller, A. Greco, D. Theriault, Z. Wu, T. Kunz, M. Betke, and J. Baillieul, "Optical flow sensing and the inverse perception problem for flying bats," in *2013 IEEE 52nd Annual Conference on Decision and Control*. IEEE, 2013, pp. 1608–1615.

[2] Z. Kong, K. Ozcimder, N. W. Fuller, and J. Baillieul, "Perception and steering control in paired bat flight," *arXiv preprint arXiv:1311.4419*, 2013.

[3] K. Sebesta and J. Baillieul, "Animal-inspired agile flight using optical flow sensing," *arXiv preprint arXiv:1203.2816*, 2012.

[4] J. Baillieul and Z. Kong, "Saliency based control in random feature networks," *arXiv preprint arXiv:1403.5462*, 2014.

[5] D. N. Lee and P. E. Reddish, "Plummeting gannets: a paradigm of ecological optics." *Nature*, vol. 293, pp. 293–294, 1981.

[6] Y. Wang and B. J. Frost, "Time to collision is signalled by neurons in the nucleus rotundus of pigeons," *Nature*, vol. 356, no. 6366, pp. 236–238, 1992.

[7] D. N. Lee *et al.*, "A theory of visual control of braking based on information about time-to-collision," *Perception*, vol. 5, no. 4, pp. 437–459, 1976.

[8] R. J. Bootsma and C. M. Craig, "Global and local contributions to the optical specification of time to contact: Observer sensitivity to composite tau," *Perception*, vol. 31, no. 8, pp. 901–924, 2002.

[9] S. Hrabar, G. Sukhatme, P. Corke, K. Usher, and J. Roberts, "Combined optic-flow and stereo-based navigation of urban canyons for a uav," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2005.* IEEE, 2005, pp. 3309–3316.

[10] J. Serres, F. Ruffier, S. Viollet, and N. Franceschini, "Toward optic flow regulation for wall-following and centring behaviours," *International Journal of Advanced Robotic Systems*, vol. 3, no. 27, pp. 147–154, 2006.

[11] E. Rondon, I. Fantoni-Coichot, A. Sanchez, and G. Sanahuja, "Optical flow-based controller for reactive and relative navigation dedicated to a four rotor rotorcraft," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2009.* IEEE, 2009, pp. 684–689.

[12] B. Oliveira, "Quadrotor stabilization using visual servoing," Master's thesis, Instituto Superior Tecnico, Universidade de Lisboa., 2013. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/395145540429/tese.pdf

[13] B. K. Horn and B. G. Schunck, "Determining optical flow," in *1981 Technical Symposium East.* International Society for Optics and Photonics, 1981, pp. 319–331.

[14] B. D. Lucas, T. Kanade *et al.*, "An iterative image registration technique with an application to stereo vision." vol. 81, 1981, pp. 674–679.

[15] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision–ECCV 2006.* Springer, 2006, pp. 430–443.

[16] ——, "Fusing points and lines for high performance tracking," in *Tenth IEEE International Conference on Computer Vision, 2005.*, vol. 2. IEEE, 2005, pp. 1508–1515.

[17] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[18] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *Computer Vision–ECCV 2010*. Springer, 2010, pp. 778–792.

[19] N. Moshtagh, N. Michael, A. Jadbabaie, and K. Daniilidis, "Vision-based, distributed control laws for motion coordination of nonholonomic robots," *IEEE Transactions on Robotics*, vol. 25, no. 4, pp. 851–860, 2009.

[20] A. Bruhn, J. Weickert, and C. Schnörr, "Lucas/kanade meets horn/schunck: Combining local and global optic flow methods," *International Journal of Computer Vision*, vol. 61, no. 3, pp. 211–231, 2005.

[21] A. Alahi, R. Ortiz, and P. Vandergheynst, "Freak: Fast retina keypoint," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2012, pp. 510–517.

[22] D. E. Shoch, "Histology of the human eye: An atlas and textbook," *The Journal of the American Medical Association*, vol. 219, no. 2, pp. 221–221, 1972.

[23] D. F. Garway-Heath, J. Caprioli, F. W. Fitzke, and R. A. Hitchings, "Scaling the hill of vision: the physiological relationship between light sensitivity and ganglion cell numbers," *Investigative Ophthalmology & Visual Science*, vol. 41, no. 7, pp. 1774–1782, 2000.

[24] M. Muja and D. G. Lowe, "Fast matching of binary features," in *2012 Ninth Conference on Computer and Robot Vision*. IEEE, 2012, pp. 404–410.

[25] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 10, pp. 1615–1630, 2005.

[26] A. Canclini, M. Cesana, A. Redondi, M. Tagliasacchi, J. Ascenso, and R. Cilla, "Evaluation of low-complexity visual feature detectors and descriptors," in *2013 18th International Conference on Digital Signal Processing*. IEEE, 2013, pp. 1–7.

[27] G. Bradski, "The opencv library (2000)," *Dr. Dobb's Journal: Software Tools for the Professional Programmer.*, vol. 25, no. 11, pp. 120, 122–125, 2000.

[28] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[29] C. S. Royden, "Mathematical analysis of motion-opponent mechanisms used in the determination of heading and depth," *Journal of the Optical Society of America*, vol. 14, no. 9, pp. 2128–2143, 1997.

[30] B. Venthur, "Python ardrone," https://github.com/venthur/python-ardrone, 2011.

J. Paul Seebacher

106 Spring Valley Rd. • Ossining, NY 10562 • (914) 588-1860
j.paul.seebacher@gmail.com

Experience

Space Exploration Technologies **Hawthorne, CA**
Propulsion Development Engineer *2014 – Present*

- Merlin Vacuum Full Thrust nozzle extension production setup.

GE Aviation **Lynn, MA**
Design Engineer - Edison Engineering Development Program *2011 – 2014*

- F414 Variable exhaust nozzle ceramic matrix composite (CMC) hardware designer and manufacturing focal.
  - Used novel CMC product line for noise reduction hardware in divergent section of F18 nozzle.
  - Created NX designs and ANSYS FEA models and substantiated through engine and vibration tests.
  - Led manufacturing introduction establishing production quality parts and reducing costs 12k$/engine set
- Passport 20 CMC Centerbody owner, certification and manufacturing focal.
  - Responsible for hardware during engine certification of commercial grade CMC exhaust hardware.
  - Design iteration to simplify manufacturing and assembly while improving quality and reducing costs.
- Real time data processing software development for Lynn engine test cells.
  - Created software to collect data from ADC units, filter bad data points, execute performance calculations and distribute, store and view data in real time.
- Engine test auto-throttle software development.
  - Produced program to automatically throttle GE38 engine mission profiles throughout certification tests.
  - Created real time data capture and plotting software for vital engine parameters and ARINC codes.

GE Aviation **Hooksett, NH**
Manufacturing Engineer - EID Coop *Summer 2010*

- Created and tested repair to achieve surface finishes on compressor blisk airfoils saving 50k$ per affected part.
- Designed robotic cell for sanding of triple-stage compressor blisks.
- Created software to check engineering tolerances against CMM data taken on airfoils, hubs, rabbets, and disks.

Dartmouth College **Hanover, NH**
Digital Electronics Lab Teaching Assistant *Spring 2011*

- Taught students to use function generators, oscilloscopes, power supplies and other lab equipment
- Tutored FPGA programming (VHDL) and debugging techniques.

Design Methodology: Senior Project *Fall 2010 âĂŞ Winter 2010*

- Designed and manufactured power electronics, controls and mechanical integration components for in-hub, front motor regenerative braking and four-wheel drive system. Implemented on a hybrid-formula race car.

Education

Boston University **Boston, MA**
M.S. Systems Engineering *2013 – 2014*

- Major: Control Theory - GPA 3.88
- Thesis: Stability and local motion planning of quadrotors using video and image processing.

General Electric **Lynn, MA**
Advanced Courses in Engineering *2011 – 2013*

- Major: Advanced Jet Engine Design - GPA 3.8
- Additional Courses: Lean Six Sigma Greenbelt, Foundations of Leadership, Fundamentals of Jet Engines

Dartmouth College **Hanover, NH**
B.A. and B.E. Engineering *2007 – 2011*

- Major: Mechanical Engineering focus in Digital Electronics - GPA 3.51

Skills & Interests

- *Fabrication Experience:* Composite prepregging and layup / Machining (Mill, Lathe, etc.) / TIG Welding MIG Welding / Mold design / Polymers / Carpentry / Plumbing / Masonry
- *Programming and CAD* Python / Matlab / NX7 / NX6 / ANSYS / Solidworks / C / VHDL / Bash / Visual Basic
- *Electronics Experience:* FPGA / Microcontrollers / Low Voltage Design and Fabrication / PCB design
- *Lab Equipment* Oscilloscope / Function Generator / Instron / Hardness Testing
- *Car Mechanics:* Tuning / Engine Rebuilds / Exhaust systems / Suspensions / Wrenching / Driving
- *Other Interests:* Baseball / Running / Biking / Weekend Projects

---

Project Experience

- *Model Rocketry:* Manufactured nosecone and telemetry system to monitor supersonic heating effects. Created rocket simulator to baseline aero design
- *Dartmouth Formula Racing Team:* Fabricated bodywork, auto clutching shifter and linkage, exhaust and cooling system. Fabricated suspension and frame and rebuilt the turbo. Team welder.
- *Digital Radio:* Implemented a digital radio on a Spartan 3A series FPGA with adaptive Morse code decoder.
- *Dorm Efficiency:* Manufactured wall mounting card reader/holder, its injection mold, its low and high voltage electronics (with RFID reading capability) and its PCB. Project reduced energy consumption by 33%.
- *Foam Bridge:* Designed and machined a foam bridge to withstand 1200 lbs. with minimum displacement.
- *Lab Polymer Testing:* Determined the effects of carbon fiber, Kevlar and fiberglass fillers in polymers.